UNIVERSITY OF CALIFORNIA, SAN DIEGO

Low-Latency Techniques for Improving System Energy Efficiency

A dissertation submitted in partial satisfaction of the
requirements for the degree of Doctor of Philosophy

in

Computer Science (Computer Engineering)

by

Richard D. Strong

Committee in charge:

Tajana S. Rosing, Chair
Dean M. Tullsen, Co-Char
Clark C. Guest
Andrew B. Kahng
George Porter
Steven Swanson

2013

Copyright

Richard D. Strong, 2013

All rights reserved.

The Dissertation of Richard D. Strong is approved and is acceptable in quality and form for publication on microfilm and electronically:

____

____

____

____

____
Co-Chair

____
Chair

University of California, San Diego

2013

DEDICATION

To all those that keep on trying.

EPIGRAPH

We are what we pretend to be,
so we must be careful what we pretend to be.

Kurt Vonnegut

TABLE OF CONTENTS

Signature Page . . . iii

Dedication . . . iv

Epigraph . . . v

Table of Contents . . . vi

List of Figures . . . ix

List of Tables . . . xii

Acknowledgements . . . xiv

Vita . . . xvii

Abstract of the Dissertation . . . xix

Chapter 1 Introduction . . . 1
- 1.1 Efficiency with Minimal Performance Impact . . . 1
- 1.2 Power Gating . . . 5
- 1.3 Fast Core Switching . . . 8
- 1.4 The Need for Efficient Network Bandwidth . . . 9

Chapter 2 Memory Access Power Gating . . . 14
- 2.1 Related Work . . . 17
- 2.2 Power Gating and Power Distribution Network . . . 20
  - 2.2.1 Programmable Power Gating Switch (PPGS) Design . . . 20
  - 2.2.2 PDN Model for Power Estimation and Circuit Analysis . . . 23
  - 2.2.3 Safe Wake-Up Mode Analysis and Equation . . . 26
  - 2.2.4 Core Wake-Up Stagger . . . 28
- 2.3 System Design . . . 30
  - 2.3.1 Centralized Wake-Up Controller (WUC) . . . 30
  - 2.3.2 Distributed, Staggered Wake Up . . . 32
  - 2.3.3 MAPG-Counter: Counter-Based Controller Design . . . 34
  - 2.3.4 TAP: Token-Based Adaptive Power Gating . . . 36
  - 2.3.5 Formal Analysis of In-Order Core Energy Savings . . . 40
  - 2.3.6 Core State Retention and Restoration . . . 42
- 2.4 Simulation Methodology . . . 44
- 2.5 Results . . . 47
  - 2.5.1 EV6 vs IO Power Gating Energy Savings . . . 48
  - 2.5.2 Execution Time and Overheads . . . 53

2.5.3 Energy Savings as a Function of Wake-up Latency 55
2.5.4 Adapting to Memory Contention 57
2.5.5 Distributed, Staggered Wake Up 59
2.6 Summary 62
2.7 Acknowledgments 63

Chapter 3 Very Fast Core Switching 65
3.1 Related Work 69
3.1.1 Thread Migration Techniques 69
3.1.2 Scheduling for Heterogeneous Multicores 70
3.2 Software Approaches to Core Switching 72
3.2.1 Modified System Calls 73
3.2.2 V1: Linux's Thread-Migration Mechanism 74
3.2.3 V2: Modified Scheduler 75
3.2.4 V3: Scheduler Fast Paths 76
3.2.5 V4: Addressing IPI Costs 77
3.2.6 V5: Cross-Core Wake Up from Quiesce 79
3.3 Simulation Environment and Workloads 80
3.3.1 Modeling Core Power Up 81
3.3.2 Workloads 82
3.3.3 Organization of Experiments 84
3.4 Microbenchmark Results 84
3.4.1 Results on Real x86 Hardware 85
3.4.2 Results on Simulated Hardware 85
3.5 Effects of Architectural Parameters 86
3.5.1 L1 Cache Sizes 87
3.5.2 Core Wake-Up Delay 88
3.6 Macrobenchmark Results 89
3.6.1 Web Benchmark 89
3.6.2 Database Benchmark 92
3.6.3 Network Streaming Benchmarks 95
3.6.4 Energy Efficiency 97
3.7 Summary 98

Chapter 4 Integrating Microsecond Circuit Switching into the Data Center 100
4.1 Related Work 104
4.2 Motivation: Reducing Network Cost via Faster Switching 106
4.2.1 Multi-layer Switching Networks 106
4.2.2 OCS Power Advantages 108
4.2.3 OCS Model 110
4.3 OCS Throughput and Latency 111
4.3.1 Throughput 112

4.3.2 Latency 112
4.4 Implementation 115
4.4.1 Mordia Prototype 116
4.4.2 Emulating TORs with Commodity Servers 121
4.4.3 All-to-All Traffic Generator 138
4.5 Evaluation 140
4.5.1 Emulated TOR Software 140
4.5.2 Throughput 142
4.6 Summary 146
4.7 Acknowledgments 146

Chapter 5 Summary 148

Bibliography 152

LIST OF FIGURES

Figure 1.1. Data center server average power and energy efficiency as a function of CPU utilization from [16] 2

Figure 1.2. Best job completion time for Hadoop sort as a function of greater network oversubscription [121].......... 10

Figure 2.1. Operation of the power-gating technique.......... 21

Figure 2.2. Wake-up current profiles with different wake-up controls.......... 21

Figure 2.3. PPGS design and inrush current profiles vs. wake-up modes 23

Figure 2.4. 16-core system power delivery network with power gating.......... 26

Figure 2.5. SPICE-calculated minimum wake-up latency for an EV6 16-core CMP with various wake-up scenarios.......... 27

Figure 2.6. Wake-up latency coefficient, T_0, as a function of PDN parameters. 29

Figure 2.7. Minimum wake-up latency as a function of wake-up stagger. 30

Figure 2.8. WUC and PPGS integration into a 4-core CMP. 31

Figure 2.9. WUC, Core-i PPGS, and Memory Subystem timing diagram.......... 31

Figure 2.10. Wake-up slot assignments with different number of slots (η).......... 33

Figure 2.11. A diagram of TAP's power-gating behavior 38

Figure 2.12. Interface for power gating and data retention. 43

Figure 2.13. TAP and MAPG-Counter EV6 energy savings 49

Figure 2.14. TAP and MAPG-Counter in-order energy savings 50

Figure 2.15. Breakdown of simulation time for each benchmark running on an EV6 core utilizing either MAPG-Counter or TAP to save energy. . 54

Figure 2.16. Energy savings for MAPG-Counter and TAP as core wake-up latency varies from 2 *ns* to 16 *ns* 56

Figure 2.17. TAP and MAPG-Counter adapting to increasing memory contention 58

Figure 2.18. Core wake-up latency improvement from stagger 61

Figure 2.19. Core energy savings improvement from stagger 61

Figure 2.20. Distributed WUC's impact on power-gated time 62

Figure 3.1. Example of a system call modified to support core switching 74

Figure 3.2. Timeline showing 2 core switches between a pair of cores, using fast-path versions of *schedule* 77

Figure 3.3. Abstract pseudo-code for modified versions of *schedule()* 78

Figure 4.1. A comparison of a scale-out FatTree network and a Hybrid electrical/optical network 102

Figure 4.2. Data center power reduction from the introduction of a core switch OCS 110

Figure 4.3. The effect of changing duty cycle on OCS bandwidth and RTT . . . 113

Figure 4.4. The effect of changing duty cycle and number of port sharers on OCS RTT 114

Figure 4.5. Logical diagram of the Mordia OCS prototype 116

Figure 4.6. The software implementation of a software TOR 123

Figure 4.7. Steps for a user program to transmit a packet. 124

Figure 4.8. Organization of day and night synchronization frames 128

Figure 4.9. A flow diagram depicting the MTOR Qdisc enqueue function . . . 130

Figure 4.10. Pseudo-code flow diagram for synchronization processing 131

Figure 4.11. A flow diagram depicting the MTOR Qdisc dequeue function . . . 132

Figure 4.12. The case for enhanced token management 134

Figure 4.13. Actual OS code for how the *enough_tokens* function stage in Figure 4.11 determines the number of tokens left 135

Figure 4.14. The impact of aging and pacing tokens on packet loss rate 136

Figure 4.15. Throughput comparison between a2a-syngen and netperf for varying MTU between two hosts 139

Figure 4.16. Host 1's Qdisc receiving UDP packets from Hosts 12–16 as it cycles through circuits connecting it to 22 other hosts. 141

Figure 4.17. Host 1's Qdisc transmitting UDP packets to Hosts 7–11 as it cycles through circuits connecting it to 22 other hosts. 142

Figure 4.18. Network throughput delivered over the OCS 143

Figure 4.19. Synchronization jitter as seen by our software TOR's OS. 145

LIST OF TABLES

Table 2.1. Estimated data for 32 *nm* HP, LOP and 22 *nm* HP, LOP in-order cores. 24

Table 2.2. Average and maximum error on the modeled wake-up time for 4-, 6-, 8-, and 16-core cases (EV6, 32 *nm* HP). 28

Table 2.3. System configuration values 45

Table 3.1. Summary of core-switching versions 72

Table 3.2. A mapping from architectural configuration name to core types for both the application and OS core 80

Table 3.3. Fraction of CPU time spent in various modes. Measurements are based on unmodified Linux on a *simulated* uniprocessor 83

Table 3.4. Microbenchmark results for *gettid* per-call delay with 1,000,000 samples per trial 85

Table 3.5. Microbenchmark results with cross-core wake up 86

Table 3.6. Effect of L1 cache size on microbenchmark results, using the V5 core-switching mechanism 87

Table 3.7. Effect of power-up delay on performance 88

Table 3.8. Simulated Web results on dual-core CPUs for 1G and 10G NICs. Values are KB transferred during 133 ms. 90

Table 3.9. Core-switch counts for 1 Web trial, dual-core X86 91

Table 3.10. Simulated Web results on quad-core CPUs. Values are KB transferred during 133 ms 91

Table 3.11. Simulated throughput for *ex_tpcb*. Values are transactions/sec. rates (for 100 transactions). This trial used 16 KB L1 caches. 92

Table 3.12. Throughput for *ex_tpcb* on dual-core X86 94

Table 3.13. Core-switch counts for 1 ex_tpcb trial, dual-core X86 94

Table 3.14. Simulated Netperf results for TCPstream. Values are KB transferred during 167 ms. 94

Table 3.15. Simulated Netperf results: TCPmaerts. Values are KB transferred during 167 ms. 95

Table 3.16. Core-switch counts for 1 netperf trial, dual-core X86 96

Table 3.17. Energy efficiency comparison between the bound and V5 kernels (KTrans/s/W or MB/s/W) 97

Table 4.1. Power consumption of data center networking components 109

ACKNOWLEDGEMENTS

I would like to take this chance to acknowledge the people, who have been instrumental in my getting a PhD. I thank my parents who instilled in me a sense of importance for continuing my education. I thank my wife, who not only withstood my nearly endless turmoils and would selflessly offer her help in my research, but who also managed to take this road to a PhD at the same time. I also would like to thank my advisors, my dissertation committee, my fellow researchers, my friends, and my family. I thank the people at the following institutions based on chronological order: J. L. Pettis VA Medical Center, UC Los Angeles, City of Hope, Uwink Inc., UC San Diego, MIPS Technologies, and HP Labs. I am grateful to Ericsson, Google, Qualcomm, Oracle, Cisco, Microsoft, CIAN, MuSyC, GSRC, and NSF grants CCF-0702349, EEC-0812072, CNS-0923523, SHF-0916127, SHF-1218666, SHF-1116667, and CCF-1162085 for supporting the research projects that contributed to my dissertation.

Chapter 2 contains material from "MAPG: Memory Access Power Gating", by Kwangok Jeong, Andrew B. Kahng, Seokhyeong Kang, Tajana S. Rosing, and Richard Strong, which appears in *Design, Automation & Test in Europe Conference & Exhibition*, 2012. The dissertation author was a principle contributor and author of this paper.

Chapter 2 also contains material from "TAP: Token-Based Adaptive Power Gating", by Andrew B. Kahng, Seokhyeong Kang, Tajana S. Rosing, and Richard Strong, which appears in the *ACM/IEEE International Symposium on Low Power Electronics and Design*, 2012. The dissertation author was a principle contributor and author of this paper. This material is copyright ©2012 by the Association for Computing Machinery, Inc. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM

must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Publications Dept., ACM, Inc., fax +1 (212) 869-0481, or permissions@acm.org.

In addition, Chapter 2 contains material from "Many-Core Token-Based Adaptive Power Gating", by Andrew B. Kahng, Seokhyeong Kang, Tajana S. Rosing, and Richard Strong, which will appear in in the *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. The dissertation author was a principle contributor and author of this paper.

Chapter 3 contains material from "Fast Switching of Threads Between Cores", by Richard Strong, Jayaram Mudigonda, Jeffrey C. Mogul, Nathan Binkert, and Dean Tullsen, which appears in *SIGOPS Operating Systems Review*, Volume 43, Issue 2 on April 2009. The dissertation author was the primary investigator and author of this paper. This material is copyright ©2009 by the Association for Computing Machinery, Inc. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Publications Dept., ACM, Inc., fax +1 (212) 869-0481, or permissions@acm.org.

Chapter 4 contains material from "Integrating Microsecond Circuit Switching into the Data Center", by George Porter, Richard Strong, Nathan Farrington, Alex Forencich, Pang-Chen Sun, Tajana S. Rosing, Yeshaiahu Fainman, George Papen, and Amin Vahdat, which will appear in the proceedings of *The Special Interest Group on*

Data Communications, 2013. The dissertation author was the secondary investigator and author of this paper.

VITA

2003	Internship J. L. Pettis VA Medical Center, Loma Linda, CA
2005	Internship City of Hope, Duarte, CA
2006	B.S. of Computer Science and Engineering University of California, Los Angeles
2006	Software Engineer Uwink Inc.
2007	Internship MIPS Technologies, Mountain View, CA
2008	Internship HP Labs, Palo Alto, CA
2009	Teaching Assistant University of California, San Diego
2009	M.S. of Computer Science University of California, San Diego
2009	Teaching Assistant University of California, San Diego
2009–2013	Research Assistant University of California, San Diego
2013	PhD of Computer Science (Computer Engineering) University of California, San Diego

PUBLICATIONS

G. Porter, R. Strong, N. Farrington, A. Forencich, P. Sun, T. Rosing, Y. Fainman, G. Papen, and A. Vahdat. Integrating Microsecond Circuit Switching into the Data Center. To appear in *Proc. Special Interest Group on Data Communications*, 2013.

A. B. Kahng, S. Kang, T. S. Rosing, and R. Strong. Many-Core Token-Based Adaptive Power Gating. To appear in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*.

A. B. Kahng, S. Kang, T. S. Rosing, and R. Strong. TAP: Token-Based Adaptive Power Gating. In *Proc. International Symposium on Low Power Electronics and Design*, pages 203-208, 2012.

K. Jeong, A. B. Kahng, S. Kang, T. S. Rosing, and R. Strong. MAPG: Memory Access Power Gating. In *Proc. Design, Automation, & Test in Europe Conference & Exhibition*, pages 1054-1059, 2012.

R. Strong, J. Mudigonda, J. C. Mogul, N. Binkert, and D. Tullsen. Fast Switching of Threads Between Cores. *SIGOPS Operating Systems Review*, 43(2):35-45, 2009.

A. K. Coskun, R. Strong, D. M. Tullsen, and T. S. Rosing. Evaluating the Impact of Job Scheduling and Power Management on Processor Lifetime for Chip Multiprocessors. In *Proc. Conference on Measurement and Modeling of Computer Systems*, pages 169-180, 2009.

S. Li, J. Ahn, R. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *Proc. International Symposium on Microarchitecture*, pages 469-480, 2009.

ABSTRACT OF THE DISSERTATION

Low-Latency Techniques for Improving System Energy Efficiency

by

Richard D. Strong

Doctor of Philosophy in Computer Science (Computer Engineering)

University of California, San Diego, 2013

Tajana S. Rosing, Chair
Dean M. Tullsen, Co-Char

U.S. data center energy consumption is expected to rise past 100 billion kWh in 2013. Approximately 50% of this energy usage can be attributed to servers, networks, and storage, and the other half goes to power and cooling infrastructures in their support. Servers are so non-energy efficient that they consume 65% of a fully utilized server's power when only 30% utilized. Even when 100% utilized, the server may not be running efficiently.

This dissertation improves the energy efficiency of data center systems in three ways. The first method improves server energy efficiency by turning off CPU cores during

long-latency memory accesses with no performance penalty to data center applications. This technique leverages peak rush current from on-chip power distribution networks to quickly charge core capacitance, and allows the core to resume execution in as little as 8.06 ns. Core state is saved through careful use of slave latches and source biasing.

A key means to increasing effective CPU utilization and energy efficiency of servers, is to leverage virtual machine and thread migration at minimal performance overhead. Our second technique speeds up software thread migration by up to 2.5× compared to Linux, with latencies as small as 933 ns. We leverage this technique to quickly migrate operating system code between asymmetric cores to reduce application energy consumption.

The techniques introduced so far assume that I/O devices have sufficient bandwidth to keep the server processor busy. In contrast to this assumption, many data centers oversubscribe their networks to reduce cost and power consumption, sometimes at the expense of overall data center efficiency. Our last contribution is a software top-of-the-rack switch capable of offloading unmodified TCP/IP traffic onto a prototype, microsecond optical circuit switch within a microsecond. We demonstrate that servers can utilize up to 95.4% of optical circuit bandwidth even when switch reconfiguration latency is reduced by three orders of magnitude to 11.5 μs, supporting the introduction of low-latency optics into the data center to radically reduce cost and power consumption of full bisection bandwidth networks.

Chapter 1

Introduction

U.S. data center energy consumption has risen steadily since the year 2000, and is expected to reach more than 100 billion kWh in 2013, costing $7.4 billion each year [5]. Further, individual data centers have increased their power demand to as much as 100 MW [34]. Data centers are expected to grow in size and number, due to demand for cloud services from mobile devices, tablets, and big data. The U.S. EPA found that 50% of data center energy usage can be attributed to servers, networks, and storage with each part accounting for 40%, 5%, and 5%, respectively [5]. Hence pressure exists to improve the energy efficiency of data centers, and such efforts must consider the contributions of servers, networks, and disks. Further, the remaining 50% of power consumption is attributed the power and cooling infrastructure that supports servers, networks and disks, and so their energy efficiency in inherently intertwined.

1.1 Efficiency with Minimal Performance Impact

Servers are one of the largest contributors to data center power. However, two disturbing trends exist in today's data center servers. First, they spend a majority of their time at 30% utilization or less [16]. This happens because data center operators over provision their data center to meet peak demands due to strict quality of service (QoS) guarantees, and reliability requirements.

For an example of strict QoS guarantees, Google search found that users who were subjected to a random delay between 100 ms to 400 ms, reduce their total number of searches by up to 0.76% [24]. Google also found that longer exposures to delay, causes an increasing reduction in searches, implying a worrisome correlation between service delay and profitability. Inside the data center, the acceptance for delay is less. Services like *Memcached* [46] and *RAMCloud* [97] can be queried numerous times by other servers to satisfy a single request, leading to sensitivity on a microsecond time scale. In addition, some services are more sensitive to the tail latency of requests, meaning that overall speed does not matter so much as the worst case. As a result, data center operators over provision their systems to avoid costly regions of operation and disgruntled users.

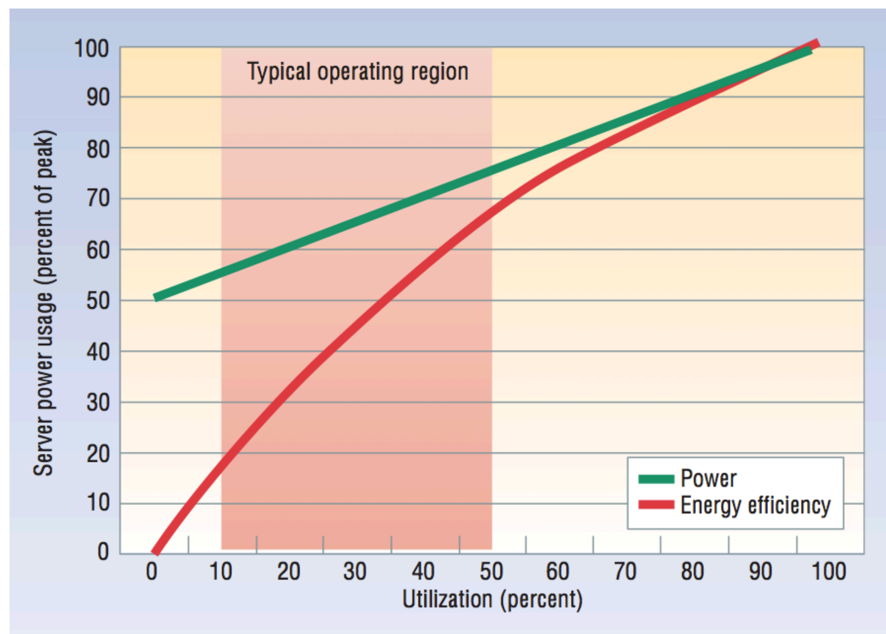


Figure 1.1. Data center server average power and energy efficiency as a function of CPU utilization from [16]

Over provisioning data center servers comes at a price since servers are not energy proportional (see Figure 1.1). An ideal energy proportional server should use power in proportion to the amount of resources it uses. However, a 30% utilized server

consumes 65% of the power of a fully utilized server [16]. Even worse, a data center that accomplishes no interesting work still consumes half as much server power as a fully utilized data center.

The lack of energy proportionality of servers is worse than the result reported in [16]. The study reports only CPU utilization, under the assumption that server utilization is proportional to CPU utilization. However, even when a server is fully utilized as measured by CPU utilization, the server many have many subcomponents, including the CPU, that are not fully utilized. For instance, the CPU may experience a long-latency memory access that misses in the cache hierarchy, causing it to stall. If the CPU executes operating system code with significant instructional dependencies, both the CPU and memory subsystem will be underutilized. Alternatively, if a network application uses the *poll* system call or non-blocking sockets to check for incoming data, but is bottlenecked by the network, then the CPU and memory will once again be underutilized. In all of these cases, the operating system would report CPU utilization at 100%. Ergo, the second disturbing trend is that a server at 100% CPU utilization may not necessarily be operating efficiently.

Moore's Law [92] may improve data center energy efficiency via smaller transistors that offer both performance improvements and power reduction. However, a transistor-based device in the data center consumes power in two forms: dynamic and leakage. Dynamic power represents the power of switching transistors while they perform useful work, while leakage power is consumed even when the device is idle. Distressingly, constraints that limit scaling of device threshold voltages, reduce device supply voltages, and improve transistor switching speed lead to leakage power becoming an increasingly dominate component of system power consumption. For example, leakage power contributes to 41% of overall server processor power to run the Spec2006 application suite on a processor built in the 22 nm technology node [78]. Transistor leakage is significant

for not only the server processor, but for many other data center devices such as memory, NICs, controllers, switches, and others.

Given that technology scaling alone will likely not solve the problem, and the critical interactions between data center software and hardware, solutions should come in the forms of both hardware and software changes. In the domain of hardware-only changes, there have been several exciting proposals. These techniques range from automatically generating custom ASICs to efficiently execute common applications [113] and advanced dynamic voltage frequency scaling (DVFS) techniques [35, 38, 63], to introducing network switches that dynamically adapt links rates to improve network energy proportionality [10]. Such hardware changes can avoid software control, allowing very fine-grained decisions for a subsystem of the data center. For the case of hardware techniques, it is important to remember that the improvement of a subsystem can have important implications for the entire data center since data centers often reuse the same components throughout their infrastructure. For example, a technique that improves the energy efficiency for a single core may be applied across millions of cores in the data center. This dissertation contributes to this area of research by investigating the use of low-overhead power gating to improve the energy efficiency of server processors.

However, software is critically involved in the interaction of data center servers. Although software can be an order of magnitude slower than hardware-only solutions, it often allows coarser-grain decisions that affect the efficiency of several data center subsystems. For instance, some publications focus on the interaction of disks, memory, and software to improve the performance of processing big data [102, 119], while others may use binary translation to more efficiently use processor resources [120]. Software changes can also yield benefits at the data center scale, as data center operators like Amazon, Rackspace, Google, and Red Hat use a virtualized infrastructure that shares hypervisors, operating systems, and services across tens of thousands of servers. A

technique that speeds up thread migration between cores, or allows virtual machines (VM) to migrate faster between servers can often be leveraged at data center scale. This dissertation develops a fast core-switching technique that can quickly migrate threads between cores on a multicore processor to enable energy efficiency opportunities that rely on constant thread migration.

The interaction of software and hardware is not limited to servers as the network is often involved in the efficiency of both. Software designs can use different software techniques, protocols (i.e. TCP and UDP), and operating system calls (e.g., *poll* and *select*), which can all change the behavior of data center network demand. Even for a fixed software configuration, the topology and offered link rates of the interconnecting switches can have dramatic impacts on the availability of bandwidth between two servers. Techniques that address these concerns range from software control of the network via OpenFlow [86] and Open vSwitch [99], to techniques that provide full bisection network bandwidth from commodity switches [12, 93]. This dissertation uses software control of the network to demonstrate the viability of integrating a prototype microsecond optical circuit switch into the data center. Optical switches offer the promise of providing full bisection bandwidth and a three orders of magnitude reduction in energy consumption. The next sections provide greater detail about the actual techniques that this dissertation proposes.

1.2 Power Gating

Up to this point, we introduce research that makes a case that servers are not energy proportional, and that transistor scaling alone will not solve the problem. Further, transistor scaling exasperates the problem due to an increased contribution of leakage power to total power. A useful example of this problem exists for server processors, which usually account for the largest contribution of server power [40]. During every

cycle that a server processor core is on, even when stalled, leakage power is consumed via gate leakage, gate-induced drain leakage, junction leakage, and subthreshold leakage.

Power gating is a technique that drastically reduces leakage power by cutting off the current path from supply to ground through introduction of a transistor switch between them. Better yet, it does not suffer from the inability to scale device threshold voltages in the face of aggressive supply voltage scaling that plagues DVFS.[1] At one end of the spectrum, functional unit power gating reduces power consumption of unused core functional units [116] with wake-up latencies of several nanoseconds. At the other end, entire cores may be power gated and woken up, with latencies of several tens of microseconds to account for saving and restoring all core state from memory [103].

Power gating cuts all supply voltage to logical cells, causing them to lose state. The techniques to address this problem differ by cell type: complex, sequential, and SRAM. Complex cells implement gate logic, and after power gating, they merely require time to power up and for their input signals to be valid. Sequential cells maintain state, and after power gating, they must power up, be reset, and restore their logical state. The fact that sequential cells must restore their logical state, means that they must save it in a modified cell that is able to survive supply voltage collapse, or that their value may be restored through an interaction of other sequential cells and complex logic. Saving sequential cells comes at roughly a 20% overhead in leakage power and area per cell [67], so the minimal subset of sequential cells should be able to retain state when power gated. Last, if SRAM were power-gated, then it too would lose state, which could potentially cause large performance penalties. Alternatively, source biasing may be used in which supply voltage is reduced by half, so that SRAM leakage is reduced, but logical state is maintained. This technique requires a separate voltage domain to supply SRAM, which

[1] DVFS is one of the most common techniques for reducing server processor power consumption in servers today

results is less leakage power savings, but also minimizes performance overhead.

Once we can power gate each cell, the next problem is when to power gate, and when to wake up from a power-gated state. Waking up from a power-gated state takes time and energy to supply all cells with the necessary charge from the supply voltage line. If the device does not remain in the power-gated state for long enough, it is quite possible that wake-up energy would exceed the energy savings from reductions in leakage power. Even if power gating can reduce leakage power consumption over power-gated intervals, if the delay to wake the device is too long, the contribution of other system components can easily increase energy consumption overall.

The goal is to power gate underutilized cores, avoid performance impact, and to improve overall server energy efficiency. Chapter 2 proposes a low-latency, power-gating technique to turn off cores stalled on a long-latency, memory access. When a core accesses a memory address that is not cached, the memory request must propagate through the cache hierarchy, chip interconnect, and get scheduled by the memory controller. If the core stalls due to the memory dependency, a controller saves the core's architectural state and then power gates the core. To avoid performance and energy overhead, the controller maintains a lower bound on each memory request's latency, and only power gates the core if sufficient time remains. This technique allows cores to make power-gating decisions at a 10 *ns* time scale without any measured performance impact. Simulations show up to 22.4% core energy savings on a memory bound benchmark. Further, avoiding performance impact allows this technique to be deployed in data center servers without impact on a service's QoS.

Power gating is best suited to address the energy efficiency of servers executing memory bound applications, but it does little to improve the efficiency of applications that experience short stalls or underutilized subcomponents in the processor. Further, the technique requires modifications to hardware which will not benefit existing servers.

1.3 Fast Core Switching

Today's servers continue to receive the benefits of the multicore era, in which a single 1 U server can support up to 64 cores [8]. As a result, the communication costs between cores have been reduced by an order of magnitude. If we consider that data centers can have tens of thousands of servers, each server has tens of cores, and that these cores are mostly underutilized, then there are, currently, millions of underutilized cores in the data center. One way to exploit this vast resource is through low-latency thread migration.

Thread migration may be used to distribute single-threaded computation between many cores to see parallel-like speedups and energy reduction due to an increase in cache capacity from many cores [66]. Low-latency thread migration can also speedup load-balancing operations to allow cores to more quickly balance the assignment of threads between symmetric cores in which maximizing simultaneous operations on a server and improving its energy efficiency [82]. For asymmetric core architectures, dynamic, fine-grained load-balancing of threads between cores can outperform static assignment between 20% and 40% on average [18]. When a server executes a system call and enters the operating system, it is possible to improve energy efficiency by switching to a simpler core due to many instructional dependencies and large working sets of operating system code [89]. Many existing servers support changing the voltage and frequency of different cores or sockets, which allows the transformation of the vast idle core resource into an asymmetric core resource.

In all of these cases, the latency of thread migration between cores is critical. Chapter 3 considers a dimension of this problem by developing a low-latency, software-based, core-switching technique that can be used to migrate applications between asymmetric cores. Asymmetric cores may be chosen to posses those resources most critical for

efficient execution of each application, enabling energy savings even when the core does not stall long enough to power gate. Specifically, the technique migrates operating system (OS) code to a simpler, energy efficient core that still offers good performance. If the OS and hardware are co-designed, then further efficiency may be achieved. Simulation and server experiments indicate that the new technique can migrate Linux OS code between cores up to 2.5× faster than previous software techniques with latencies as small as 933 ns. Use of core switching with asymmetric cores can improve server processor energy efficiency by up to 6.16× for OS intensive codes. Data center servers may leverage this low-latency technique to improve the efficiency of load-balancing VMs[2], thermal management, and computation spreading with minimal impact to QoS agreements that demand responses from critical servers within 10s of microseconds or less.

1.4 The Need for Efficient Network Bandwidth

Both power gating and fast core switching can offer improvements in server energy efficiency through hardware-only and software/hardware co-design. However, they both operate under the assumption that servers have access to the data they need to process locally, or that their I/O devices have sufficient bandwidth to keep the processor busy. In contrast to this assumption, sometimes a server or VM spends its time waiting for I/O, especially from the network due to oversubscription.

Data centers commonly oversubscribe their network at the higher levels of the switching fabric in order to manage cost, complexity, and power consumption of the network. As a result, should two or more servers need to communicate through the higher levels of the network, they may only receive a fraction of their requisite bandwidth demand called the *oversubscription ratio*. The Cisco design manual suggests that an oversubscription ratio between 2.5:1 and 8:1 is common in large cluster designs [1].

[2]A VM is often a thread of some operating system

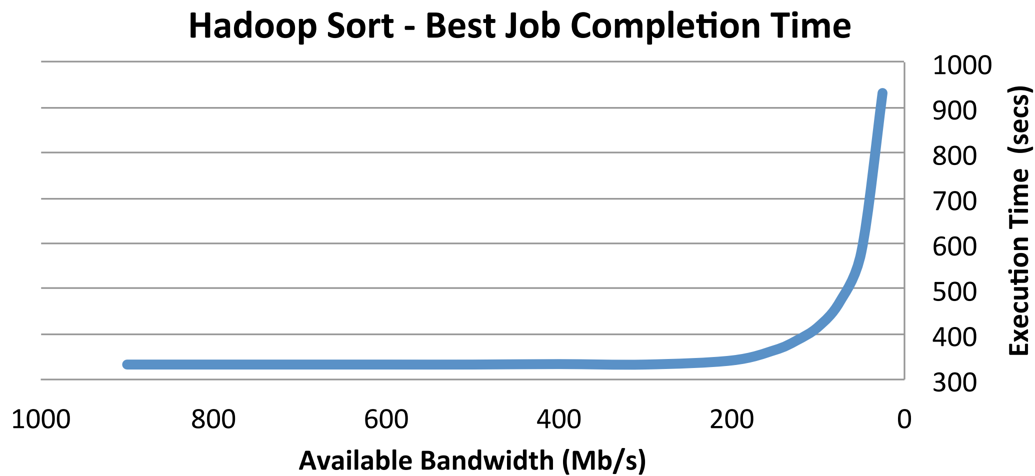


Figure 1.2. Best job completion time for Hadoop sort as a function of greater network oversubscription [121].

Other data center centers claim to suffer from an oversubscription ratio of 240:1 for 1 Gb/s servers [49]. Such oversubscribed networks may account for only 5% of data center power, but they can dramatically slow down virtualized services and nearly double server energy consumption to complete the same amount of work [121] (see Figure 1.2). At the extreme, some services like Hadoop [4] kill and retry parallel computations across hundreds of servers if a sufficient number of servers suffer from poor network behavior, resulting in high energy and performance overhead.

Many commodity servers often come integrated with 10 Gb/s network devices, and 40 Gb/s adapters compatible with PCI Express are available for $260. Servers can offer these high-bandwidth network devices data from memory buffers at a data rate limited by either memory bandwidth[3] or PCI Express[4]. Today, some big data applications use server memory as a distributed parallel cache to save recently generated data for map reduce like jobs that require several iterations over that data, yielding order of magnitude improvements in performance over a disk only solution [119]. RAMCloud uses DRAM

[3] 79.55 GB/s for an 8-core, 2.9 GHz, Intel E5-2690 server running STREAM [84]
[4] PCI Express v4.0 can offer 31.51 GB/s bandwidth across 16 lanes [25]

as a drop in replacement for disk to offer up to three orders of magnitude improvement in storage performance [97].

Even the underlying storage for the data center is poised to greatly increase in speed. Today, data centers use spinning disk technologies that have been around for decades, and operate near 7,200 RPM. Flash-based SSDs are commodity, scale linearly in performance in RAID arrays, and can offer sequential bandwidths of up to 569 MB/s. Phase change memory could potentially offer an order of magnitude improvement in sequential throughput at 2.932 GB/s [26]. Such an increase in raw storage bandwidth for a distributed file system may put significant pressure on the network. Taking these factors into account, network bandwidth demand will likely continue to rise in the foreseeble future.

To prevent the negative consequences of oversubscription and increasing network bandwidth demand, data centers can introduce a scaled-out network that offers full bisection bandwidth between any two hosts [12, 93]. However, such a network needs to be provisioned for the worst case communication pattern, leading to as many as nine layers in the largest networks. In general, an N-level scaled-out network built from k-radix switches can support $k^N/2^{N-1}$ servers, with each layer of switching requiring $k^{N-1}/2^{N-2}$ switches (though layer N itself requires half this amount). The implication is that higher level switches become less likely to be utilized and waste energy, as they also are not energy proportional [10]. Further, a full bisection network would consume as much power as the servers, if the servers became energy proportional and were 15% utilized on average [10].

In summary, today's data center networks contribute to only 5% of data center power, but offer bandwidth at an oversubscription ratio, which can adversely affect the energy efficiency of large distributed applications. Underlying storage technology, and software techniques that utilize memory for storage are poised to increase network

bandwidth demand of commodity servers. Left unchecked, the network, especially near the core switch level, acts as a bottleneck against improving data center energy efficiency. Current techniques to provide networks with full bisection bandwidth can remove this bottleneck, but their complexity and power consumption may offset any gains seen from energy proportional servers. Thus, improvements in server energy proportionality should be followed by improvements in network energy consumption and offered bandwidth.

Electrical packet switch (EPS) networks have long been a cornerstone in data center networks, because of their improvements in offered bandwidth and ability to schedule each switch on a per packet basis. Today, 10 Gb/s Ethernet, high-radix switches with up to 96 ports are commodity. However, port switches with 40 Gb/s often have much lower port counts in the range of 16 to 24 ports, which adds complexity to the design of the data center network. Should the port counts of a 40 Gb/s and 100 Gb/s EPS continue to scale, then the limits of skin effect and Ethernet cable input power for practical data center distances will likely impede efforts to offer bandwidth in excess of 100 Gb/s [87].

In contrast, an optical fiber suffers loss of only $2 * 10^{-4}$ dB/m [87], separating the concern of transmission rate from distance for today's data center sizes. Each fiber of an optical network can use wave division multiplexing to transmit between 44 - 177 channels (depending on channel spacing) at 20 Gb/s [71] per channel yielding between 880 - 3540 Gb/s. Optical circuit switches (OCS) based on 3D-MEMS Glimmerglass can offer switches that scale at 240 mW/port, agnostic to data rate compared to an electrical packet switch that consumes as much as 14.1 W/port at 10 Gb/s [43]. Such an OCS is also a space switch capable of each port delivering all frequencies to a specific output port.

There are at least two major challenges in integrating an OCS into the data center. First, an OCS port does not process each individual packet to determine a destination port. Instead, a circuit is setup between a source and destination port, which means that

network control resides closer to the server. Second, in order to switch between circuits, a reconfiguration time is necessary. During this time, mirrors move into position, stop, vibrate, and no data may transit through the OCS port. A reconfiguration time imposes a duty cycle on the offered bandwidth of the switch, and also increases the average delay of packet through the switch if it shares a source or destination port with some other circuit.

Therefore, EPS networks face several challenges in terms of scaling up in bandwidth, while still offering full bisection bandwidth between hosts. Optics offers an opportunity to increase bandwidth per port by an order magnitude or more, while also allowing a two orders of magnitude reduction in power per port. However, an OCS offers its bandwidth in the form a circuit which could affect latency of packets through the network.

Chapter 4 explores the introduction of a microsecond optical switch (OCS) into the data center, with the focus on reducing the monetary cost and power consumption of providing full bisection network bandwidth between all servers. Specifically, this chapter designs a software top-of-the-rack switch (TOR) capable of delivering unmodified TCP/IP network traffic to a prototype OCS that can establish new circuits in as little as 11.5 μs. The TOR is composed from a modified, network card device driver and OS that mutually schedule each other at a microsecond granularity. This technique enables OCS experiments with commodity servers and unmodified applications. Experiments with a small server cluster indicate that today's systems can utilize up to 95.4% of OCS bandwidth, giving rise to data center networks with full bisection bandwidth at a 24.7% reduction in data center power compared to an electrical technology. Consequently, data center servers that do not suffer network bottlenecks run more efficiently, and are more likely to meet their QoS requirements.

Chapter 2

Memory Access Power Gating

Chapter 1 explains that servers contribute to 40% of the power consumption of data centers, the processor is often the largest contributor to this consumption, and that even a 100% utilized processor may not be executing efficiently. Indeed, during every cycle that a server processor core is on, even when stalled, leakage power is consumed via gate leakage, gate-induced drain leakage, junction leakage, and subthreshold leakage. A core may stall quite often if it is intensely accessing the memory subsystem, as every time a thread makes a memory request that misses in the L1 cache, the core is subjected to a variable access latency. This variable latency often translates into a core stall during which no forward thread progress occurs and energy is wasted. For a 32*nm* out-of-order EV6 core, stall energy can be up to 39.1% of total energy consumption for the Spec2006 benchmarks [78]. Further, many data center applications spend significant time in the OS, executing system calls and interrupts, which suffer from a large working set and memory bound behavior.[1]

Previous work has reduced core energy waste by lowering core frequency and voltage (DVFS) for memory-intensive threads when directed by L2-cache misses [37, 38, 63]. Some schemes can even direct core DVFS behavior based on signals from the L2-cache and estimates of instruction-level parallelism [77]. A slower core frequency results in

[1] For more details, refer to Chapter 3, Table 3.3

fewer cycles waiting for the memory subsystem. Scaling down both frequency and voltage results in an estimated cubic dynamic power and quadratic leakage power savings [62]. However, the inability to scale device threshold voltages, coupled with aggressive scaling of supply voltages (subject to overdrive and performance requirements), means that cores have little room to reduce voltage during DVFS [105]. The net effect is decreased energy savings from DVFS, which motivates the development of new techniques to reduce core energy consumption while waiting for the memory subsystem.

Power gating is a technique that drastically reduces leakage power by cutting off the current path from supply to ground through introduction of a transistor switch between them. At fine granularity, functional unit power gating reduces power consumption of unused core functional units [116] with wake-up latencies of several nanoseconds. At coarse granularity, entire cores may be power gated and woken up, with latencies of several tens of microseconds to account for saving and restoring all core state from memory [103]. An intermediate mechanism that we design provides the ability to power gate an entire core, wake up a power-gated core in about 10 *ns*, and maintain the core's architectural and cache state. It uses a combination of a *programmable power gating switch* (PPGS), state retention cells, and source biasing to enable the core to efficiently enter and exit a power-gated state.

In this chapter, we compare two architectural techniques for directing the PPGS's behavior: *Memory Access Power Gating Counter* (MAPG-Counter) and *Token-Based Adaptive Power Gating* (TAP). MAPG-Counter directs core power gating by tracking the duration of core-stall periods with a counter for those stall periods that last longer than a last-level cache miss. It uses core-stall history to predict future stall periods. With these predictions, MAPG-Counter can predict the duration of stall periods, and direct the PPGS when to power gate a core with minimal performance hit. The benefit of this technique is in both its simplicity and its significant energy savings for in-order

cores. However, without detailed knowledge of all core memory requests, out-of-order execution and hard-to-predict stalls become a significant barrier to the use of this power gating mechanism.

TAP deterministically applies power gating during core stalls which are caused by the variable latency of requests to the memory subsystem. TAP achieves this by providing the capability to track every ongoing memory request and the expected response time for each memory access that misses in the L1 cache. An expected lower bound on latency is sent to each core's PPGS by modifying the cache controllers to send a token on any miss where the token includes an estimate of the access latency of a next-level memory hit. The result is that TAP can support power gating with no measured performance loss.[2]

To summarize, this chapter makes the following contributions:

- We compare two practical techniques, MAPG-Counter and TAP, to direct the PPGS controller on unmodified applications. We show that TAP can offer 2.58× the average energy savings of MAPG-Counter for out-of-order cores.
- We show that MAPG-Counter can achieve slightly higher energy savings on average than TAP for an in-order core.[3]
- We show that TAP has no measurable impact on application performance or QoS.
- We formally analyze TAP's energy savings for in-order cores to achieve predictions of energy savings for an arbitrary memory hierarchy and application, with 0.82% and 9.75% average and maximum error.
- We compute the time at which a power-gating action breaks even with the wake-up energy to be 8.53 *ns* and 17.17 *ns* for in-order and out-of-order cores, respectively.

[2]Our technique could cause a performance hit if a token got lost on the on-chip interconnect, or if a token got significantly delayed. However, our simulations do not observe this behavior.

[3]The HP Moonshot System released in 2013 contains 45 hot pluggable servers, that are based on the Intel Atom S1260, a in-order core at 2 GHz.

- We decompose MAPG-Counter and TAP behavior into time spent power gating, waking up the core, restoring core state, and overhead to show that core wake-up and restore time averages 1.9% of execution time.
- We demonstrate that both MAPG-Counter and TAP can adapt to an increase in memory contention by increasing power-gated time by 2.02× and 3.69×, respectively, as the number of threads increases from 1 to 32.
- We design and implement a staggered wake-up scheme capable of reducing wake-up latency up to 58.2%; this results in a 3.14% increase in energy savings for TAP.

2.1 Related Work

Power gating has been studied at both architectural and circuit levels. Microarchitectural works typically examine the questions related to use of different power-gating modes, what to power gate, predicting when to power gate, and control algorithms to avoid energy penalties from poor power-gating decisions. Circuit-level papers typically analyze different circuit techniques aimed at reducing wake-up latency, efficiently retaining logic states, minimizing ground bounce, and achieving resilience to process variation. The following briefly reviews representative works in these two areas.

Hu et al. [58] propose power gating as a technique to reduce functional unit leakage power when applications underutilize their functional units. Specifically, they power gate the floating-point and fixed-point units according to three different predictors which are respectively ideal, time-based, and branch-misprediction-guided. The best technique (branch-misprediction-guided) is able to put functional units to sleep for up to 40% of total cycles with only 2% performance loss. The authors of [58] also develop equations to estimate the break-even points for power gating an out-of-order superscalar

processor. However, although they build a power consumption model with precise analysis of virtual supply voltage during power gating, they do not consider the wake-up energy required to restore circuit nodes.

Lungu et al. [83] show that in many cases, the predictor of [58] can lead to increased energy consumption. A monitor that controls the use of power gating is introduced to bound the performance and energy penalty for misbehaved applications. Madan et al. [14] extend the idea of Lungu et al. to the core level, and propose a "guard mechanism" that reduces harmful use of power gating.

Power-gating technology is also readily visible in leading commercial products. The recent Nehalem architecture employs power gating at the core level to reduce leakage power on idle cores, but 100 *ms* is required to wake up a core [73, 76]. AMD [103] has improved this power-gating technique by optimizing the wake-up sequence to skip built-in self tests (BIST) and restoration of cache state; this results in wake-up times as short as 75 μs. In today's systems, the OS typically power gates the cores in the idle loop, missing out on power gating long memory accesses.

In the realm of circuit innovation, the pioneering work of Horiguchi et al. [56] has been followed by many works on fundamental circuit design issues related to power gating, including switch-cell sizing, data-retention methods, physical-implementation methodologies, and mode-transition noise analysis and reduction. The recent survey of Shin et al. [106] gives an excellent summary of the history and highlights of power-gating techniques.

We propose a *multi-mode power-gating* technique that allows multiple wake-up modes to minimize wake-up latency. Configurable power gating has been used in the past to mitigate process variation, reduce ground bounce noise, and minimize wake-up time. Agarwal et al. [11] and Singh et al. [107] examine multiple sleep modes that feature different wake-up overheads and leakage power savings. Use of multiple sleep

modes achieves an extra 17% reduction in leakage power compared to a single power-gating mode. Also, one of the sleep modes can reduce leakage power by 19% while preserving circuit state. However, these energy savings are based on static traces of bus activity and do not address the runtime problem of predicting *when* to power gate. In addition, the reported results are likely optimistic since wake-up noise and the overhead of implementing low-voltage sleep control signal distribution are not considered.

To minimize ground bounce during mode transition, Kim et al. [69] control turn-on voltage (V_{GS}), which makes sleep transistors turn on in a non-uniform stepwise manner. Kim et al. [70] propose a tri-mode power-gating structure in which a PMOS switch is combined in parallel with traditional NMOS power-gating switches. The additional PMOS transistor supports intermediate power-saving state-retaining modes at low-supply voltage, and reduces ground bounce noise during transitions between normal and power-gated modes. Chowdhury et al. [32] propose a similar tri-mode (i.e., RUN, HOLD, CUT-OFF) power-gating technique using PMOS switches in parallel with NMOS footer switches, combined with additional NMOS switches in parallel with PMOS header switches. Kim et al. [68] propose a programmable-width power-gating switch that adjusts the widths of power-gating switches to compensate for core-to-core process variation occurring in multicore systems. Finally, Zhang et al. [122] propose a multi-mode power-gating technique using three NMOS switches with different sizes and threshold voltages. Using various combinations of the three switches, they can provide multiple power-gating modes with different leakage savings. They also note that their method is tolerant to process variation.

The most similar work to this chapter, Memory Access Aware Power Gating for MPSoCs [81], examines the potential to power gate an in-order core while monitoring a single memory bus and estimating memory latencies. A controller that sits at the memory bus sends explicit commands to each core to power gate and to wake up from

a power-gated state. The controller estimates memory latencies by tracking whether each memory request is a row buffer hit or miss. However, this work does not consider out-of-order execution, and is limited in scalability to a system with a single memory bus, which precludes understanding of its application to data-center servers. In addition, it does not consider the importance of core location and state information for determining safe wake-up modes, the possibility of using staggered wake-up to reduce the latency of core wake-up from a power-gated state, or the scalability of their designs. By contrast, this chapter addresses these issues, is applicable to out-of-order cores, formally analyzes the energy savings for in-order cores, considers the importance of core location and wake-up stagger, and considers the scalability of the design to many-core processors.

2.2 Power Gating and Power Distribution Network

This section provides a low-level analysis of our power-gating methodology and its impact on the power distribution network. Section 2.2.1 gives the details of the programmable power-gating switch, Section 2.2.2 describes our models for capacitance of a core and voltage noise in the power distribution network, Section 2.2.3 explains how we model core wake-up mode constraints and the benefit of a *staggered* wake-up.

2.2.1 Programmable Power Gating Switch (PPGS) Design

As noted above, power gating cuts off leakage current paths between supply (Vdd_core) and ground (Vss) by using switch transistors (often, high-V_{th} or long-channel devices). A typical power-gating methodology with *header* switches is illustrated in Figure 2.1. When the pg_enable signal goes low, the header switches turn off and leakage current is reduced. While in the power-gated state, all logic gates connected to the virtual supply (Vdd_int) lose their logical states. Setting the pg_enable signal to high resumes circuit operation after a delay that corresponds to charging circuit capacitive

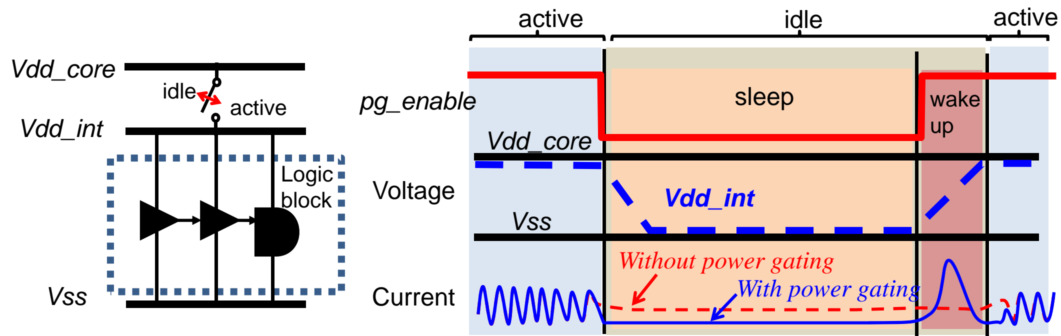


Figure 2.1. Operation of the power-gating technique.

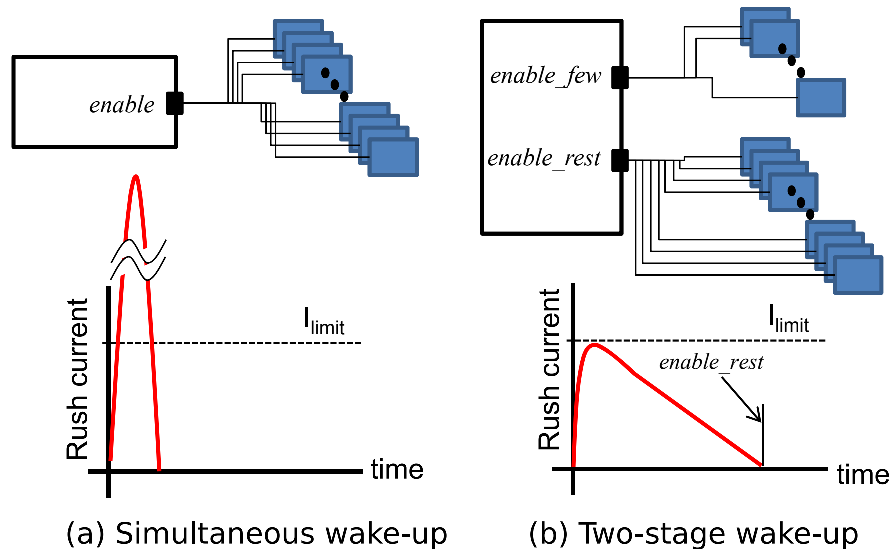


Figure 2.2. Wake-up current profiles with different wake-up controls.

loads, resetting memory elements, and restoring state from retention flip-flops connected to Vdd_core.

The delay to charge circuit capacitive elements is a function of total design charge (Q) and peak charging current (I_{limit}). If all header switches turn on simultaneously, a large "inrush" current charges internal nodes in minimal time. To satisfy inrush current upper limits (too-large IR drop can affect functionality of neighboring active blocks), header switches are partially turned on in sequence, which increases charging time to at least $T_{charge} = Q/I_{limit}$. Minimal charging time is achieved with a rectangular current

profile, but such a profile requires very fine-grained control of header switches. To avoid this design complexity, we use a two-stage wake-up control [51] where the first stage (*enable_few* signal) turns on header switches to allow I_{limit} charge current. The remaining header switches are turned on in the second stage (*enable_rest* signal) once the circuit nodes are nearly charged, resulting in a triangular charging current profile (see Figure 2.2b). This increases the wake-up latency to at least twice the minimum square wake-up profile, but simplifies signal connections.

To maximize opportunities for power gating subject to wake-up inrush current and supply noise constraints, we seek to enable multiple *wake-up modes*, with a range of wake-up latencies, per core. Figure 2.3 shows our *programmable power-gating switch* (PPGS) for a core, along with the wake-up current profile for different wake-up modes. We configure the number of first-stage wake-up switches to control the inrush current as shown in Figure 2.3b. With the dynamic configuration of the PPGS, we can minimize the wake-up time according to the core configurations — e.g., the number or location of active cores relative to the waking-up cores. To power gate a core, all mode selection signals $m[0-9]$ are set to one, which turns off all switches at the same time.[4]

Core wake-up time and inrush current are determined by the mode selection. For example, Mode 1, which has the longest wake-up time and smallest inrush current, is set by $m[0] = 0$ and $m[1-9] = 1$. Thus, $m[0]$ is enabled by signal *enable_few* and $m[1-9]$ is enabled by signal *enable_rest*. Mode 2 is set by $m[0-1] = 0$ and $m[2-9] = 1$; inrush current increases with the number of first-stage switches, while wake-up time decreases, as shown in Figure 2.3b. The other modes can be set similarly.

[4]Due to the large resistance of off-state switches, inrush current from simultaneous turn off is negligibly small compared to wake-up inrush current.

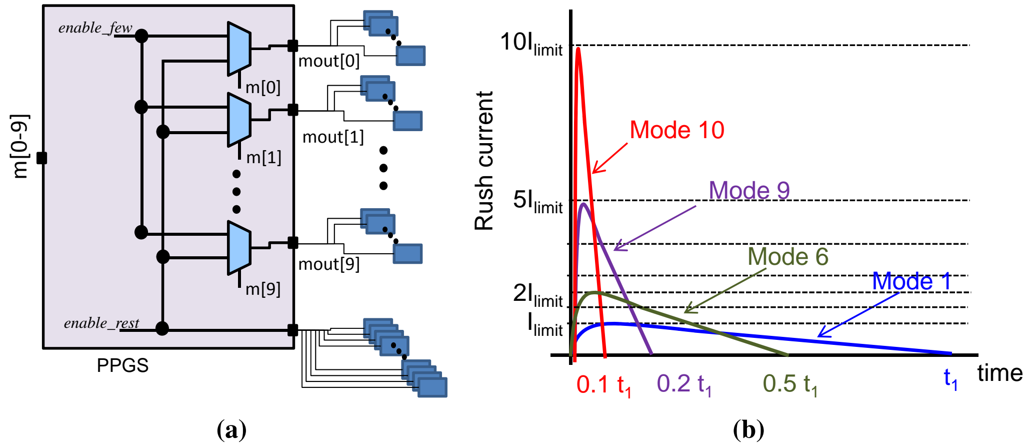


Figure 2.3. PPGS design and inrush current profiles vs. wake-up modes

2.2.2 PDN Model for Power Estimation and Circuit Analysis

Table 2.1 shows estimated design parameters, power-gating results and PDN-model parameters for 32 *nm* and 22 *nm* in-order cores with high performance (HP) and low-operating power (LOP) devices. To study wake-up latency and inrush current, we estimate the total charge for core logic and interconnect capacitance as $Q_{core} = (C_{logic} + C_{int})Vdd_core$, where Q_{core}, C_{logic}, and C_{int} represent total charge, device capacitance, and interconnect capacitance for a single core without caches. We estimate a core's total transistor count using [78] to determine the core's area and average transistor density. Based on this transistor count and parameters from the 2009-2010 *International Technology Roadmap for Semiconductors* (ITRS) [6], we estimate C_{logic} and C_{int}. The inrush current limit (I_{limit}) and on-current (I_{active}) are estimated from McPAT data for peak power and average power, respectively.

From the calculated charge (Q_{core}), the minimum wake-up latency with a rectangular form current profile is $T_{min-charge} = Q_{core}/I_{limit}$, and the minimum two-stage wake-up latency (Figure 2.2b) is $2 \times T_{min-charge}$.

We estimate leakage power consumption during power gating of the core logic

Table 2.1. Estimated data for 32 *nm* HP, LOP and 22 *nm* HP, LOP in-order cores.

Estimated Data	32 *nm* HP	32 *nm* LOP	22 *nm* HP	22 *nm* LOP
Design Data				
Vdd_core (V)	1.00	0.77	1.00	0.77
core area (mm^2)	4.593	4.608	2.701	3.657
logic area (mm^2)	2.891	2.863	1.635	1.636
C_{core} (F)	7.53E-9	7.48E-9	4.58E-9	4.58E-9
total charge (C)	7.53E-9	5.76E-9	4.26E-9	3.30E-9
core leakage (W)	0.355	0.042	0.147	0.019
I_{active} (A)	0.725	0.374	0.371	0.233
I_{limit} (A)	1.298	0.674	0.701	0.632
Power Gating and Wake-up				
$T_{min-charge}$ (ns)	5.08	7.36	6.40	6.55
wake-up energy (pJ)	3.30E+3	1.91E+3	2.24E+3	1.60E+3
# head switches	9,664	6,222	5,516	5,127
leakage in PG state (W)	8.03E-3	7.14E-4	3.37E-3	3.59E-4
leakage reduction in PG	97.74%	98.29%	97.71%	98.12%
PDN Model				
# bump	45	45	95	95
R_{shared} (Ω)	0.01	0.01	0.01	0.01
$L_{pkg-core}$ (nH)	7.69E-4	7.76E-4	6.44E-4	6.44E-4
$R_{pkg-core}$ (Ω)	1.54E-5	1.55E-5	1.29E-5	1.29E-5
C_{decap} (F)	1.51E-9	1.50E-9	9.16E-10	9.16E-10
R_{PDN} (Ω)	0.07	0.10	0.12	0.15

and SRAM, as follows. For the core logic, leakage from retention registers and header switches must be taken into consideration. We assume that (live-slave type) retention flip-flops have 20% more leakage power than normal flip-flops during power gating [67]. For SRAM, we assume that the (separate) SRAM supply voltage is scaled using source biasing, and we estimate leakage based on [101].

Following the methodology of previous works [51, 59, 69], we construct a detailed PDN model that includes package parasitics to enable realistic noise analysis under various wake-up scenarios. Power is delivered from an external voltage regulator module

(VRM) through a printed circuit board (PCB), a package ball, package interconnect, microbumps, on-die redistribution layers, the on-chip PDN, and power-gating switches. We model the entire power delivery network including power-gating switches as a simplified RLC circuit as shown in Figure 2.4. Package inductance and series resistance from VRM to bumps for a core are lumped as in-series inductance and resistance.[5] The PDN in package shared by multiple cores is represented as a resistance mesh with a branch resistance of R_{shared}. There are three variant models depending on the state of the core — core in active mode, core being woken up, and core in sleep mode (see Figure 2.4). On-chip decoupling capacitance C_{decap} is assumed to be 20% of C_{core} as in Huang et al. [59].

PDN parameter values in Table 2.1 are from personal communication with industry experts [39] and reflect production designs at the 28 nm foundry half-node. Bump density is assumed to be 45 bumps per mm^2, and the number of bumps is then calculated from logic area (I/O signals are peripherally located in the SoC die plan). The package inductance and resistance to a bump are respectively assumed to be $0.05nH$ and $1m\Omega$ based on empirical data. The lumped package inductance $L_{pkg-core}$ and resistance $R_{pkg-core}$ for a single core are respectively calculated as L_{pkg}/N_{bump} and R_{pkg}/N_{bump}, where N_{bump} is the number of bumps.

We measure the Vdd_core and Vdd_int voltages of all cores using HSPICE. We vary the number of cores being woken up, and search over all configurations of woken-up and active cores. For each configuration, we find the minimum wake-up latency that satisfies two IR drop constraints: (a) Vdd_int of active cores should drop by no more than 5% and (b) Vdd_core of standby cores should drop by no more than 40% so as to retain data in retention circuits [39].

[5] Note that we do not model inductance of the on-chip power mesh. High-frequency effects are not relevant to the wake-up current analysis, and wire dimensions are such that resistive impedance dominates. To our knowledge, our approach matches that used in advanced SoC signoff methodologies today.

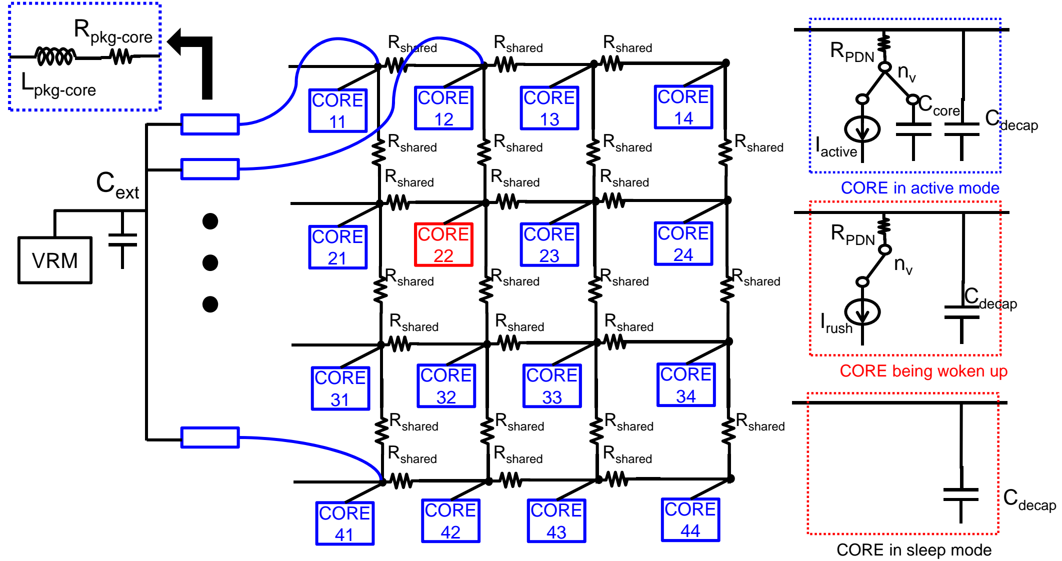


Figure 2.4. 16-core system power delivery network with power gating.

2.2.3 Safe Wake-Up Mode Analysis and Equation

It is possible to select PPGS wake-up modes based on the worst-case wake-up time for each *number* of idle cores. However, the worst-case wake-up time assumption limits the benefit of power gating. A core's minimum wake-up time is constrained by the voltage noise seen by neighboring active cores – in particular, some *critical* active neighbor core where the voltage noise constraint is first violated. The voltage noise of an active core is mainly affected by adjacent woken-up cores and the latencies (i.e., associated inrush currents) with which they wake up. In other words, we may exploit knowledge of core locations to reduce pessimism. We have developed a model that determines the minimum wake-up time based on the number and location of active and woken-up cores. To simplify the model, we assume that all woken-up cores have the same (uniform) wake-up latency, but in principle our methodology easily extends to non-uniform cores and wake-up latencies.

Figure 2.5 shows the minimum wake-up time according to the location and status of cores for an example case of an EV6 16-core CMP. In the figures, *A* denotes the critical

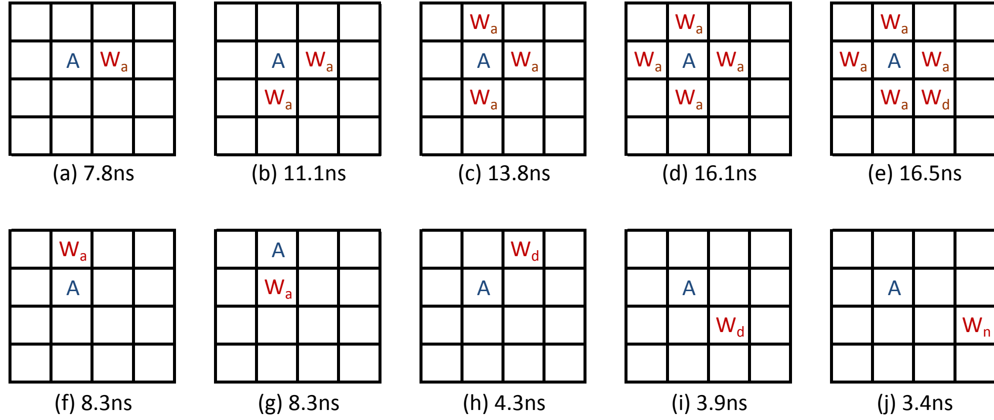


Figure 2.5. SPICE-calculated minimum wake-up latency for an EV6 16-core CMP with various wake-up scenarios.

active core, W_a are adjacent woken-up cores, W_d are diagonally adjacent woken-up cores, W_n are non-adjacent woken-up cores, and blank squares are idle or non-critical active cores. The wake-up latency increases approximately as the square root of the number of adjacent woken-up cores (Figure 2.5 (a) - (e)). Woken-up cores in the diagonal adjacent (W_d) or non-adjacent positions impact wake-up latency less than adjacent woken-up cores (Figure 2.5 (f) and (g)). Cores located at an edge position (Figure 2.5 (h)) experience increased minimum wake-up latency.

From such observations, we model the minimum wake-up latency based on the core status at each location as:

$$T = T_0(w + \beta \cdot x + \gamma \cdot y + \delta \cdot z)^{\alpha} \tag{2.1}$$

where T_0, α, β, γ and δ are fitting coefficients, w is the number of adjacent woken-up cores, x is the number of diagonally adjacent woken-up cores, y is the number of other (non-adjacent) woken-up cores, and z is the number of active or adjacent woken-up cores located at the edge.

Table 2.2. Average and maximum error on the modeled wake-up time for 4-, 6-, 8-, and 16-core cases (EV6, 32 *nm* HP).

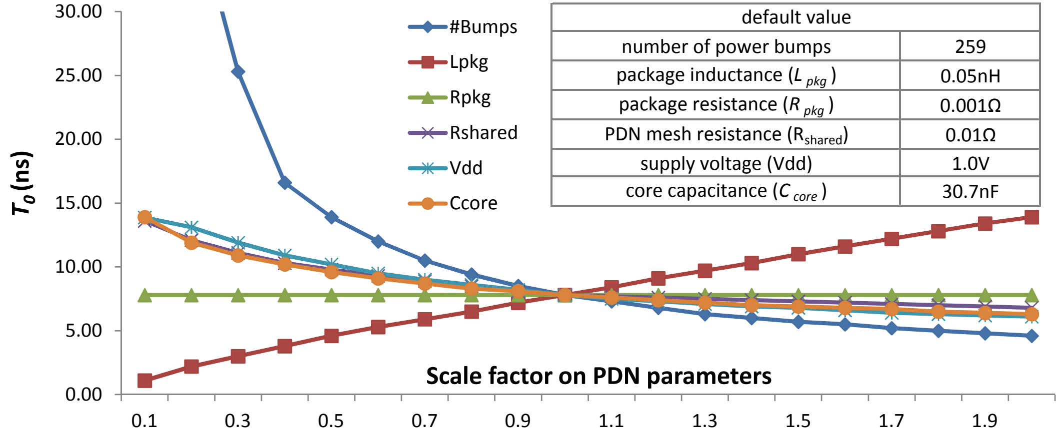
core	coefficient					error	
	T_0	α	β	γ	δ	average (%)	maximum (*ns*)
4-core	7.9	0.50	0.35	0.15	0.15	2.64	0.37
6-core	7.9	0.50	0.35	0.15	0.13	1.93	1.10
8-core	7.9	0.50	0.30	0.15	0.13	2.31	1.65
16-core	7.9	0.50	0.20	0.10	0.10	1.57	1.40

We have verified our model with SPICE and model the wake-up times for 4-, 6-, 8-, and 16-core CMPs for all location permutations. Table 2.2 shows the results. Our model has an average error of 2.64%, 1.93%, 2.31% and 1.57% for 4-, 6-, 8-, and 16-core CMP cases, respectively.

The results of the SPICE simulation have varying degrees of sensitivity to power distribution network (PDN) parameters - the number of bumps, package inductance (L_{pkg}), package resistance (R_{pkg}), PDN mesh resistance (R_{shared}), supply voltage and core capacitance. We have assessed the minimum wake-up time sensitivity to variations in the PDN model. Figure 2.6 shows the change in the T_0 coefficient when each PDN parameter is scaled from 0.1× to 2× (a 20× range!) with respect to our default values, which are obtained from personal communication with industry experts. Since the actual wake-up latency depends on PDN variations, the T_0 coefficient will be determined by testing the actual packaged chip. It is important to note that our conclusions regarding energy savings and overheads remain qualitatively the same across the range of PDN parameter values – i.e., our conclusions are quite robust to the PDN design choices.

2.2.4 Core Wake-Up Stagger

In the above wake-up analysis, we assume that all cores could wake up simultaneously which is the worst case. However, wake-up latency is significantly reduced when



default value	
number of power bumps	259
package inductance (L_{pkg})	0.05nH
package resistance (R_{pkg})	0.001Ω
PDN mesh resistance (R_{shared})	0.01Ω
supply voltage (Vdd)	1.0V
core capacitance (C_{core})	30.7nF

Figure 2.6. Wake-up latency coefficient, T_0, as a function of PDN parameters.

we *stagger* the wake-up sequence so that two cores wake up at slightly different times (e.g., offset by 1 *ns*). We design the wake-up controller to insert *stagger* between waking cores to reduce wake-up latency. Figure 2.7 shows minimum wake-up latency for an EV6 16-core CMP when we add stagger between woken-up cores. The minimum wake-up time (y-axis) is reported for the worst case for each number of woken-up cores (x-axis). When stagger is zero, wake-up time increases according to the number of woken-up cores. However, if we avoid simultaneous wake-up, minimum wake-up time reduces greatly. When two, three and four cores are waking up within an interval of three cycles (0.9 *ns*), we obtain 18.8%, 31.9% and 40.3% wake-up latency reductions, respectively, over simultaneous wake-up. From SPICE results in Figure 2.7, we can see that the minimum wake-up time does not increase with staggered wake up when the number of woken-up cores is larger than four. We model the minimum wake-up time with Equation (2.1) for up to three woken-up cores by changing the parameter α from Table 2.2. The dotted lines in Figure 2.7 show the modeled wake-up latency from Equation (2.1) and its error with respect to SPICE simulation. Our measurements of model accuracy show an average (maximum) error of 2.66% (7.62%), 1.89% (6.61%), 0.93% (3.59%) and 2.51% (3.08%) for the 4-, 6-, 8-, and 16-core CMP cases, respectively.

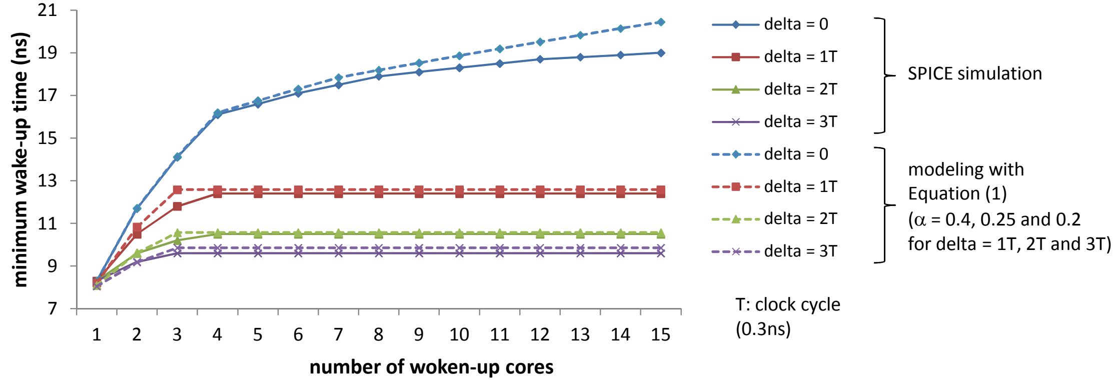


Figure 2.7. Minimum wake-up latency as a function of wake-up stagger.

2.3 System Design

We now present our architectural modifications used to control power gating for each core. A memory access power-gating controller must provide three functions. First, the controller ensures that each core's PPGS uses a wake-up mode that does not violate supply voltage noise constraints of the system when waking up a core. Second, the controller should be able to predict the expected duration of core stalls. Last, the controller must retain essential core architectural and performance related state. Together, these functions allow for energy savings and minimal performance hit without violating voltage noise constraints. The rest of this section describes how TAP and MAPG-Counter provide these three functions.

2.3.1 Centralized Wake-Up Controller (WUC)

The WUC is a centrally located wake-up controller (see Figure 2.8) that listens in on the cache interconnect and orchestrates the assignment of wake-up modes to core PPGSs. The WUC maintains a lookup table that maps the possible values of variables w, x, y and z from Equation (2.1) to safe wake-up modes. For the 16-core case, the WUC requires 12*1000 bits of (SRAM) registers to hold all entries. In addition, the WUC maintains the status of each core (idle, active, power gated, or waking-up) to determine

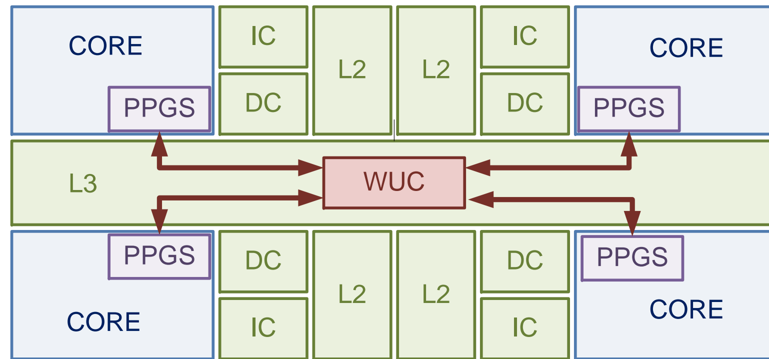


Figure 2.8. WUC and PPGS integration into a 4-core CMP.

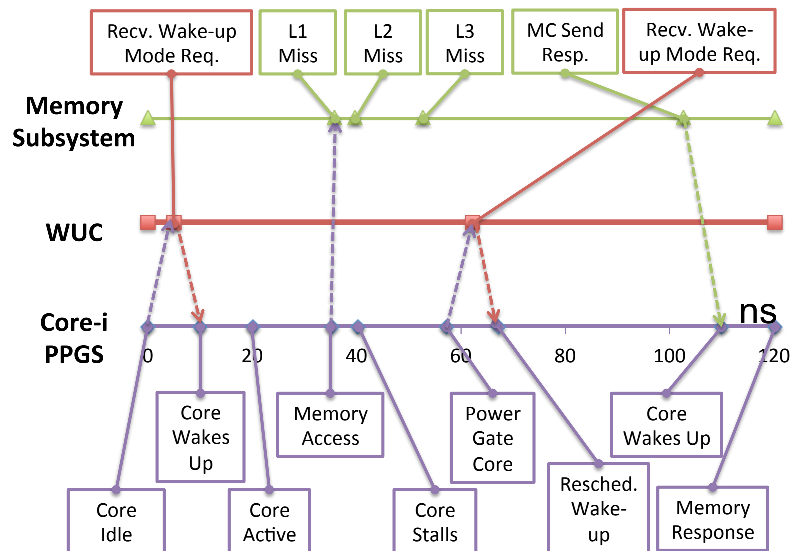


Figure 2.9. WUC, Core-i PPGS, and Memory Subystem timing diagram.

the entry to lookup for a new request.

Figure 2.9 shows how a core wakes from an idle state, power gates during a stall, and then wakes up again via communication with the WUC. At time 0 *ns*, the core is idle and power-gated off. The core wakes up by its PPGS requesting a worst-case wake-up mode that it may assume is always safe to use (provided it notifies the WUC). At 5 *ns*, the WUC receives a request, looks up a safe wake-up mode in its table based on the system state, and returns that mode to the core PPGS. The PPGS wakes up the core and the core executes code. At time 35.5 *ns*, the core attempts to access the memory subsystem which causes a stall at 40.5 *ns*. At 57.5 *ns*, the core PPGS detects a core stall dependent on a memory miss and then power gates the core. At the same time, the PPGS requests a lower-latency wake-up mode from the WUC in hopes of power gating for longer. The WUC receives this wake-up mode request at 62.5 *ns* causing the WUC to update the state of the core. Should there be no conflicting wake-ups, the WUC may issue a one-use lower-latency wake-up mode to the requesting core. The core PPGS receives this response at 67.5 *ns* and may reschedule the wake-up time of the core. The PPGS wakes up its core at 110 *ns* for the memory response at 120.5 *ns*.

2.3.2 Distributed, Staggered Wake Up

The last section designs a Wake-up Controller (WUC) for the worst case when all cores wake up simultaneously. However, wake-up latency is significantly reduced when we *stagger* the wake-up sequence so that two cores wake up at slightly different times (e.g., offset by 1 *ns*). In such a case, stagger reduces the worst-case peak current and voltage noise that a core may experience. We now consider how to integrate stagger into a many-core design; analysis of the benefit of stagger in given in Section 2.5.5.

To use staggered wake up, a controller must give wake-up modes and times to each core. For a large multicore system (e.g., 64 cores), a given core's PPGS may not

tolerate the latency to communicate with a centralized WUC due to propagation and queuing delay across the chip. However, we observe that non-adjacent cores do not significantly affect core wake-up latency, and with proper guardband, only adjacent cores (eight cores at most) need be considered.

This observation motivates a distributed design which assigns each core to a recurring wake-up slot. In this scheme, each core is given a recurring slot at which it can start waking up. To avoid any performance hit, a core should select a slot before the deadline to start waking up. The average wake-up delay for a core is defined by two degrees of freedom: the number of unique wake-up slots, η, and the stagger between two adjacent wake-up slots, ψ. The worst-case reduction in power-gated time occurs when a core predicts that it would need to wake up ε seconds before its assigned slot such that $\varepsilon < \psi$, causing the core to wake up $\eta * \psi - \varepsilon$ seconds earlier. Given a core that wakes up at any time, uniformly at random, the average expected reduction in power-gated time is $\frac{\eta * \psi}{2}$.

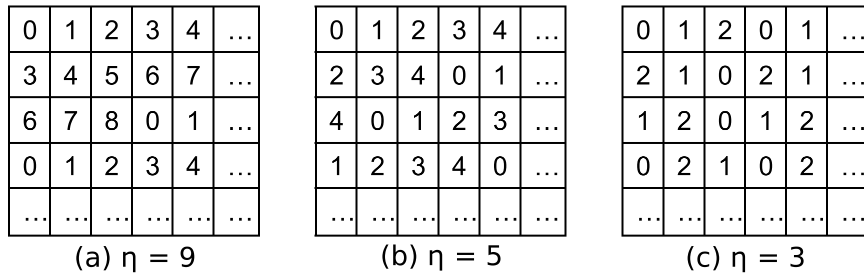


Figure 2.10. Wake-up slot assignments with different number of slots (η).

Values of η and ψ should be chosen to maximize a core's power-gated time and minimize the safe wake-up latency of the core. Given η, wake-up slots should be assigned to cores to minimize the number of adjacent woken-up cores. Figure 2.10 shows three ways of assigning wake-up modes to cores such that the number of adjacent woken-up cores is minimized with a preference given to cores waking up simultaneously

in the diagonal position. An increase in η and ψ reduces the maximum number of simultaneous core wake-ups and the minimum safe wake-up latency. At the same time, increasing these two parameters increases average expected reduction in power-gated time. According to our simulations, system energy savings are maximized when η and ψ equal 5 (no simultaneous wake-ups) and 0.9 *ns*, respectively. This setting results in a 10.3 *ns* minimal wake-up latency and 2.25 *ns* $\pm$ 1.30 *ns* average reduction in power-gated time per core power-gating opportunity, compared to 9.6 *ns* with 0.9 *ns* stagger for an ideal centralized WUC, when simulated on GEM5 [20] with the Spec2006 benchmarks (see Section 2.5.5).

2.3.3 MAPG-Counter: Counter-Based Controller Design

In this subsection, we consider how MAPG-Counter predicts core-stall duration, adapts to steady changes in average core-stall duration, and quickly scales down the duration of predicted stall windows if it over estimates. We can calculate the interval over which a core must be power gated so as to break even on wake-up energy costs from Table 2.3.[6] To avoid both energy and performance overhead, a core must receive no memory response for a time greater than the break even time plus the delay to wake up a core. A core-stall period longer than the sum of these two events yields energy savings. We find for our system described in Table 2.3 that there are energy saving opportunities when the last-level cache misses and memory is accessed. Last-level cache accesses, on the other hand, are too short to get any energy saving opportunities.

Performance overhead is avoided by (i) predicting a long stall interval and (ii) waking the core for the expected end of the stall interval. The long stall interval is predicted by counting the number of stalled cycles after a core memory access. If the number of cycles reaches beyond the expected latency at which the core would have

[6] In our $Energy_{wake-up}$ calculation, we ignore ripple effects from waking up cores since the effective operating voltage is small during the wake-up.

received a last-level cache hit response (22 *ns* according to Table 2.3), the PPGS calculates the expected arrival time of the memory response and power gates the core if sufficient time will elapse to save energy. The expected arrival time is estimated as the memory's row-buffer miss latency (45 *ns*) plus a value, δ. We compute δ as an exponential moving average of the difference between the actual and expected response arrival times, except when the expected arrival time is greater than actual, then δ is immediately set to the difference. The actual algorithm is outlined below:

MAPG-Counter Prediction Algorithm:

$diff \leftarrow MemDelay_{actual} - MemDelay_{expected}$

if $diff < 0$ **then**

$\delta \leftarrow \delta + diff$

else

$\delta \leftarrow \alpha \cdot \delta + (1-\alpha) \cdot diff$

end if

Quickly adapting to overprediction avoids repeated late core wake-ups and performance hits that would otherwise have to wait for the exponential moving average. In the common case, the exponential moving average will adapt to variable memory latency caused by contention for the memory resource, optimizing the duration of the power-gating window.

To provide this functionality, the MAPG-Counter controller is located within the PPGS of Figure 2.8. Whenever the core stalls, the MAPG-Counter controller performs two actions. First, it times the duration of the core stall in terms of clock cycles by interfacing with the core-stall signal and clock. This action allows the controller to create a history of core-stall durations that it may use to train Algorithm 2.3.3. Second, after a number of stall cycles greater than a last-level cache hit response time occurs, it must provide a prediction of the duration of the core stall duration. The PPGS uses this

prediction plus the wake-up mode from the WUC to decide if a power-gating opportunity would save energy.

To implement MAPG-Counter, each core requires a 15-bit register to save δ. The 15-bit register can track latencies as large as *3.28* μs (twice the longest stall latency observed for 32 threads) with a granularity of $100ps$. The input to the δ register is controlled with a MUX that selects between one of two inputs as defined in the if/else clause of Algorithm 2.3.3. Without modification, Algorithm 2.3.3 requires at least one multiplier as a part of the else condition which could result in significant power expenditure depending on technology. This requirement can be removed by setting α to 0.75 and using a combination of shifters and adders. Each core also requires a counter that starts once the core stalls; this is realized as a 15-bit counter to match the accuracy of the prediction scheme. To orchestrate these two registers per core, a controller is required that (i) takes as input the pipeline-stall signal, the clock, and the counter, and (ii) outputs the PPGS control control signals and the write control signals for the δ register. Overall, only 30 bits of register storage are required per core.

2.3.4 TAP: Token-Based Adaptive Power Gating

TAP informs each PPGS about expected memory latency by modifying the cache controllers to send tokens on cache misses that include an estimate of the lower-bound access latency of a next-level memory hit derived from Table 2.3 and a timestamp of creation[7]. The controllers send the tokens to the PPGS of the core that requested the memory access. Once the PPGS receives the token, it looks at the lower-bound latency to satisfy the request and power gates the core if the core is both stalled, and idle long enough to save energy. Should the core receive more than one token for simultaneous memory requests, it will track each expected response separately and schedule the resumption

[7]A cache controller sitting on the core side of a shared NUCA cache would require a per-bank lower-bound access latency

of core execution to satisfy the earliest response. If a token is delayed in the memory subsystem by a controller or queue, the PPGS can compare its arrival time with its generation timestamp and previous tokens to determine whether the token should be ignored.

Whenever a memory request misses all the way to the memory controller, the response latency experiences a significant amount of variability. This variability is caused by the complexity of DRAM memory [123], which includes bank queues, availability of the data in the row-buffer, writing wrong address row-buffers, accessing the column in the row-buffer, and channel contention between banks. TAP adapts to memory variability by adding a special token. As soon as the last-level cache experiences a miss, a token is sent to the requesting core's PPGS with an estimated completion time of *UNKNOWN*. This is a directive to the PPGS to start power gating its core immediately and to expect one additional token with the ETA of the memory response. Once the memory controller submits the memory access to one of the banks and determines whether the access is a row-buffer hit or miss, it sends the second *ETA* token to the core's PPGS with the ETA of the response assuming that there is no memory channel contention. The PPGS receives the second token before the response and schedules core wake-up for the appropriate time.

Figure 2.11 shows a timing-accurate diagram of a PPGS power gating the core in response to messages from the TAP technique. At time 0 *ns*, a memory request occurs that will miss in the cache hierarchy and cause a memory access. The PPGS then receives tokens for the L1, L2 and L3 misses. Just after receiving the L2 token, the core stalls due to a dependency. After the L3 token is received, the PPGS decides to power gate the core and saves all core state. The core is then power gated and the memory controller (MC) sends an updated ETA for the memory response. At 70 *ns*, the PPGS begins waking up the core. At 78 *ns*, the core state is restored and the pipeline is restarted. The memory

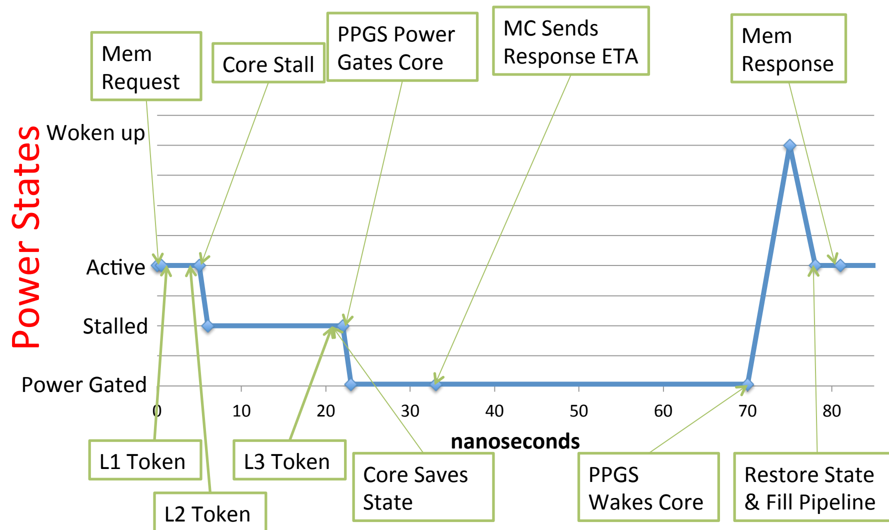


Figure 2.11. Diagram depicting the core going through various power states (Power Gated, Stalled, Active, Woken up) as the PPGS power gates the core on a memory access.

response comes back at 81 *ns* and the core resumes execution as if nothing happened.

The benefit of TAP is that core-level power gating can be directed by system-level information about the memory subsystem. This information represents lower-bound estimates of when a memory response can arrive. Because TAP operates on lower-bound estimates, it avoids over-prediction of core idle latency and achieves *zero* performance impact. The disadvantage of lower-bound estimates is that TAP misses out on potential power-gating time and additional energy savings.

To implement TAP in hardware, additional structures are added to both the memory controller and the core. For each bank of each rank of memory, we add a 15-bit delay counter that indicates the soonest time at which the bank would go idle. Similar to the MAPG-Counter above, it is capable of tracking up to 3.28 μs at a granularity of 100s of picoseconds. With each cycle, every memory controller decrements its counter by the number of picoseconds in a memory clock cycle until the counter reaches a minimum of zero. When the memory controller schedules a memory operation to a particular bank, it increments the bank's counter with the lower-bound estimate for the completion of

the memory operation. If the memory operation is a row-buffer hit, the bank counter is incremented by the time to issue the command, perform the column address select, and transfer the data across the memory bus. If the memory operation is a row-buffer miss, then the counter is incremented by the time to issue the command, pre-charge the row-buffer, issue the row lookup, perform the column address select on that row, and finally transfer the bytes across the memory bus. The value of the counter is the ETA returned to the core's PPGS by the token sent from the memory controller. To quickly determine a row-buffer hit, a register at each bank maintains the row-buffer address of the last memory access.

Each core also requires additional state to track currently valid tokens. For each unique address memory request that causes a token-generation event, we require 80 bits of storage. The first bit indicates validity of the entry. The next 64 bits contain the physical address of the request. The last 15 bits track the lower-bound ETA for the given request. In the worst case, we require sufficient entries to track the maximum number of parallel memory requests with unique physical cache line addresses that can issue from a core. For our technique, this number is limited by the number of MSHR (Miss Status Handler Registers) queue entries in the instruction and data caches, which is 20 for our EV6 architecture and 4 for our in-order architecture. However, our simulations show that fewer entries are actually required because long core stalls do not usually occur with a large number of parallel memory requests. For example, the benchmark *astar*, which has little benefit from our techniques, does experience 20 parallel requests while the benchmarks *mcf* and *gobmk*, which benefit the most from our techniques, experience at most 13 parallel memory requests. In any case, we estimate for the worst case that the support to track tokens at the core adds 1456 μm^2 of area overhead per core (0.05% of EV6 area), while the modifications to estimate memory latencies add 677 μm^2 area overhead per memory rank (0.02% of EV6 area).

2.3.5 Formal Analysis of In-Order Core Energy Savings

We now derive the expected energy savings from TAP for an in-order core, to gain intuition regarding how energy savings change with system conditions, and to independently verify our reported energy savings. In the following terms and equations, latencies are in units of seconds, power is in units of watts, and energy is in units of joules. When a core experiences a level$_i$ cache miss, it receives either a token or response from the level$_{i+1}$ cache. The idle period between when the core receives the level$_i$ token and the level$_{i+1}$ token or response can be estimated by Equation (2.2), where $L_{tx:L_i \rightarrow L_{i+1}}$ is a bus (transmit) latency from level$_i$ to level$_{i+1}$ cache miss, and $L_{hit:L_i}$ is a memory hit latency at level$_i$. Packets from both the level$_i$ and level$_{i+1}$ caches need to travel through the same memory hierarchy from level$_i$ upwards to the core. The only difference is that the packet from the level$_i$ cache travels twice across the interconnect between level$_i$ and level$_{i+1}$, waits for the level$_{i+1}$ cache controller, and waits twice for the level$_i$ cache controller.

$$L_{\Delta miss:L_i} = L_{tx:L_i \rightarrow L_{i+1}} + L_{hit:L_{i+1}} + L_{tx:L_{i+1} \rightarrow L_i} + L_{hit:L_i} \quad (2.2)$$

Because of core wake-up latency, waking the core when it receives the token or response from the level$_i$ cache would incur a significant performance penalty, as this effectively increases the level$_i$ cache miss latency by the core wake-up latency. We avoid this performance overhead by preemptively waking up the core even if there is a miss in the level$_{i+1}$ cache. This reduces the period of energy savings in some cases, but avoids the performance overhead. Further, core wake-up costs energy, which places a constraint on how long the idle period must last to amortize the energy loss from core wake-up.

The core's wake-up event is not the only overhead. When a cache sends a token, it must contend for the CPU-side ports of each cache on the way to the core, traverse the shared interconnect, and wait in any queues to shared resources. These delays can reduce the period over which our technique power gates the core. In summary, we can estimate the energy savings of power gating a $level_i$ cache miss using Equation (2.3). $E_{miss:L_i}$ is the energy savings on a $level_i$ cache miss; $L_{miss:L_i}$ is the latency to propagate a request from the core to the $level_{i+1}$ cache and back; $L_{token:L_i}$ is the latency to propagate a request from the core to the $level_i$ cache and send a token back to the core; L_{core_wakeup} is the core wake-up latency; $P_{leakage}$ is the core leakage power; R is the factor of reduction of leakage from power gating; and E_{core_wakeup} is the energy to wake up a core from the power-gated state.

$$\begin{aligned} E_{miss:L_i} = & (L_{miss:L_i} - L_{token:L_i} - L_{core_wakeup}) \\ & \cdot R \cdot P_{leakage} + E_{core_wakeup} \\ & + L_{core_wakeup} \cdot P_{leakage} \end{aligned} \tag{2.3}$$

We extend this analysis to estimate the energy savings from token-based power gating for all levels in the cache hierarchy in Equation (2.5). $E_{save:L_N}$ denotes the energy savings for an N-level cache hierarchy; $T_{perc_idle_from_miss:L_i}$ is the percent of time the core spent idle waiting for a $level_i$ cache miss (estimated as $M_{L_i} \cdot L_{\Delta miss:L_i}$); $E_{miss:L_i}$ is defined in Equation (2.3); M_{L_i} is the number of $level_i$ cache misses per second; and P_{total} is the total active power. The denominator is simply the sum of idle and active energy for when no power gating is being used. The numerator is equal to core energy during a $level_i$ cache miss without power gating, minus the energy with power gating summed across all cache levels: i.e., the sum of energy savings for each cache level. The total

system time is factored from the numerator and denominator, leaving the percentage of time spent in active execution and waiting for level$_i$ cache misses. We compare energy savings measured from McPAT [78] and M5 with the analytical model shown in Equation (2.5) and see a good match. The average error between M5 and the equation is 0.82% with a maximum error of 9.75% for *lbm*.[8] This model demonstrates that TAP's energy savings is a strong function of core wake-up latency and memory hierarchy.

$$\kappa = T_{perc_idle_from_miss:Li} \cdot P_{leakage} \tag{2.4}$$

$$E_{save:L_N} = \frac{\sum_{i=1}^{N}(\kappa - E_{miss:Li} \cdot M_{Li})}{\sum_{i=1}^{N}(\kappa) + \frac{(1 - \sum_{i=1}^{N}(T_{perc_idle_from_miss:Li})) \cdot P_{miss:Li}}{P_{leakage}/P_{total}}} \tag{2.5}$$

2.3.6 Core State Retention and Restoration

To avoid losing core state that is required for correct and efficient execution, essential sequential and SRAM cells must be retained. We replace a subset of sequential cells with live-slave retention flip-flops [67] which can be triggered to retain their logical values before a power-gating action at a cost of 20% increase in area and power versus a normal flip-flop. Only those sequential cells comprising the architectural registers

[8]Our model overestimates the energy savings for *lbm* because it assumes that all memory accesses will be row-buffer misses. This assumption works well for the Spec2006 benchmarks except *lbm*, because *lbm* has sufficient locality in its memory accesses to experience row-buffer hits. If the memory system uses a closed page policy, where all accesses are row-buffer misses, then our assumption would in fact be correct. However, our current result compares against an open page policy to measure how much error exists in our simple model for the hard to estimate case.

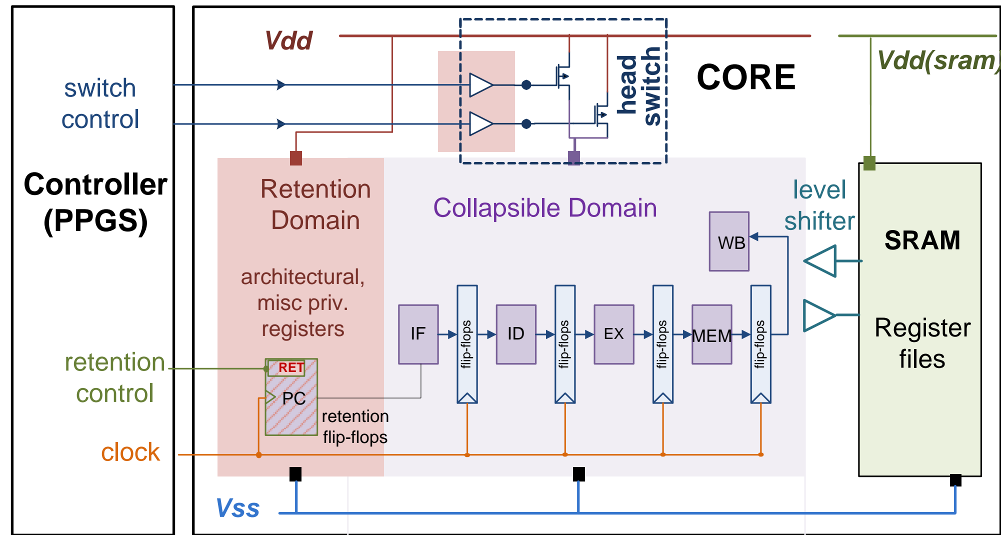


Figure 2.12. Interface for power gating and data retention.

necessary to refill the pipeline are selected, which results in 3.4% area overhead for the processor. SRAM cells are retained through source biasing [101] in which the supply voltage is reduced to 50% of nominal supply voltage so that SRAM leakage is reduced, but logical state is maintained. This technique allows for saving the contents of L1 caches, TLBs, branch predictor state, physical registers, etc. To provide supply power during power gating, a separate non-collapsible voltage domain provides power to the retention flip-flops and SRAM cells. Thus, as the power is gated from combinational logic and non-essential sequential cells, the separate voltage rail provides power to maintain core state. The overhead from multi-power domains and separate voltage rails already exist for power-gating cores today. Figure 2.12 shows an in-order core implementation for power gating and restoration with retention flip-flops.

Additional cycles are required for the power gating and wake-up sequence, and to account for the time to disable/enable the clock, trigger data retention, refill the pipeline, and de-assert/assert the clamps. We model the entire power down and wake-up sequence as in [65]. For example, to wake-up an EV6 core after signals *enable_few* and *enable_*

rest have charged core logic, it takes 1 cycle to enable the clock signal, 1 cycle to asynchronously reset logic, 1 cycle to restore registers from retention flip/flops, and 7 cycles to restore the pipeline which takes 3.03 *ns* at 3.3 *GHz*.

2.4 Simulation Methodology

Table 2.3 summarizes all system parameters in our experiments. The system has 4 cores, each with its own private L1 and L2 caches, and a large shared L3 cache. The L3 cache forwards requests to the memory controller through a shared memory bus. The L1 and L2 cache configurations are 32 *KB*-8way and 256 *KB*-8way. The L3 cache is a relatively large, 8*MB*-16way, which we expect to minimize pressure on the memory subsystem and hence minimize gains we see from our power-gating technique. We model an out-of-order core, the DEC-Alpha EV6, clocked at 3.3 *GHz* and able to issue six instructions on each cycle. We also model an in-order, dual-issue, DEC-Alpha EV4 core, clocked at 2.0 GHz.

We simulate the system with the GEM5 simulator [20]. GEM5 is a full system simulator that can boot an unmodified OS. It features cycle-level models of an out-of-order core, the cache hierarchy, and the interconnect. We integrate GEM5 with DRAMSim2 [104] to provide cycle-level modeling of the memory subsystem including the memory controller, DRAM modules, and shared channels used for communication. We modify GEM5 to support our power-gating methodology described in Section 2.3. We simulate our system with 21 of the Spec2006 benchmarks using the Simpoint methodology [98] in which 100M-instruction representative regions of execution are determined for each benchmark. To simulate each region, we fast-forward to 100*M* instructions before the region, warm-up the memory and caches, and perform detailed simulation.

Once simulation is complete, we enter the system configuration and performance counters to McPAT [78] to model power consumption. McPAT is comprised of a power,

Table 2.3. System configuration values

parameter	value	notes
IO core model	DEC-Alpha EV4	
IO core clock	2.0 GHz – 1.2 GHz	
IO execution	2-way, in-order	
IO functional units	2 ALU, 1 IMULT 1 FPALU	
EV6 core model	DEC-Alpha EV6	
EV6 core clock	3.3 GHz – 1.9 GHz	
EV6 execution	6-way, out-of-order	
EV6 functional units	6 ALU, 2 IMULT 2 FPALU	
ICache/Dcache	32 KB-8way, 1 cycle	
L2 Cache	256 KB-8way, 4 ns	Private per core
L3 Cache	8 MB-16way, 13 ns	Shared
Memory	DDR3, 2GB, 50 ns	
Core-to-L1 token latency	0.5 ns	controller delays
Core-to-L2 token latency	4.5 ns	controller delays
Core-to-L3 token latency	17.5 ns	controller delays
Core-to-WUC latency	5 ns	controller delays
PPGS wake-up modes	4.5 ns – 16.9 ns	SPICE
IO pipeline refill latency	2 ns	4-pipeline stages
IO core wake-up energy (IWE)	4,020 pJ	Charge cells
IO leakage power (ILP)	0.486 Watts	McPAT [78]
IO PG leakage reduction (ILPR)	97.74%	[67]
IO PG break even point	8.53 ns	$IEW/(ILPR*ILP)$
IO DFLT core wake-up latency	8.06 ns	SPICE
IO FUPG wake-up energy	1780 pJ	McPAT, ITRS [6]
IO FUPG wake-up latency	6.0 ns	SPICE
EV6 pipeline refill latency	2.12 ns	7 pipeline stages
EV6 core wake-up energy (EWE)	15,358 pJ	Charge cells
EV6 leakage power (ELP)	0.916 Watts	McPAT [78]
EV6 PG leakage reduction (ELPR)	97.65%	[67]
EV6 PG break even point	17.17 ns	$EWE/(ELPR*ELP)$
EV6 DFLT core wake-up latency	10.2 ns	SPICE
EV6 FUPG wake-up energy	9641 pJ	McPAT, ITRS [6]
EV6 FUPG wake-up latency	6.4 ns	SPICE

area, and timing framework that provides off-line power and area estimates for full systems designed in technology nodes between 90 *nm* and 16 *nm*. McPAT generates values for dynamic power, leakage power, peak power, thermal design power, and area. We update McPAT's *technology.cc* file to accurately reflect the ITRS 2010 update report [6].

We compare both techniques to dynamic voltage and frequency scaling (DVFS) via simulation. We calibrate our DVFS settings to match those of [30] for the 32 *nm* technology node, in which a 7.5% reduction in voltage follows each 20% reduction in frequency. To direct the DVFS policy, we apply the technique from [38], which uses a cycle-per-instruction based metric, μmean, to detect memory bounded phases of execution. During execution, we sample the application's μmean to determine the most aggressive DVFS setting that may be used to save energy while sustaining at most a 5% performance hit. In addition, we also consider an *oracle* DVFS technique that chooses the DVFS point that results in the lowest energy-delay product (EDP). This technique takes an arbitrary performance hit as long as more energy is saved. For both policies, we model the availability of five DVFS modes which include 100%, 95%, 90%, 80%, and 60% frequency.

In addition, we compare the two systems to Functional Unit Power Gating (FUPG) [14, 58, 83]. In this scheme, a controller monitors the core's functional units for stall periods that last at least 5 *ns*, and then power gates the functional unit when such a stall occurs. Meanwhile, the controller also monitors the duration of the power-gating period to ensure that over every 10,000 *ns* interval, at most 2% performance overhead occurs. Should a functional unit power gate at a performance loss of up to 2% (meaning that the functional unit continually wakes up later than it is needed), then the controller disables power gating for the remainder of the 10,000 *ns* interval. At the end of the interval, the controller resets and attempts to power gate functional units again. The wake-up latency

and energy for FUPG are given in Table 2.3.

Unless otherwise stated, our results assume a 32 *nm* HP circuit technology, core wake-up latencies from Table 2.3, and a four-core system that is 50% utilized (two cores idle), which results in conservative energy savings during little memory contention, shorter core-stall durations, and utilization of a slower wake-up mode than if only one core was utilized. The following overheads were considered when modeling the reported results (including the Oracle policy): core wake-up energy, core wake-up delay, core pipeline-refill latency, retention overhead of live-slave retention cells, SRAM leakage during source biasing mode of operation, and voltage noise safety.

2.5 Results

We now analyze MAPG-Counter and TAP to understand their energy savings and how those energy savings depend on system configuration and utilization. Also, we show that both techniques' potential to save energy improves as core wake-up latency reduces. We make the case that TAP is more adaptable to system variability and usually achieves higher energy savings than MAPG-Counter and functional unit power gating. Last, we reveal that staggered wake-ups are an easy way of reducing core wake-up latencies in CMPs. In the following, Section 2.5.1 examines the energy savings of the *two techniques*, MAPG-Counter and TAP for both EV6 and IO CMPS. The remaining subsections focus solely on the EV6 CMP, because there is little difference in the analysis between the EV6 and IO CMPs. Sections 2.5.2 dissects simulation time of the two techniques into time spent power gating, waking up, restoring state, executing code, and adding execution overhead. Sections 2.5.3 and 2.5.4 examine energy savings as a function of wake-up latency and memory congestion. Last, Section 2.5.5 shows the benefit of staggered wake-up on energy savings for a CMP with up to 16 cores.

2.5.1 EV6 vs IO Power Gating Energy Savings

Figures 2.13 and 2.14 compare the energy savings of TAP with the energy savings of an oracle memory predictor (Oracle), MAPG-Counter, FUPG [14, 58, 83], DVFS-Oracle, and DVFS-μmean [38] for an EV6 CMP and IO CMP, respectively. The discussion that follows focuses on the EV6 CMP, but highlights IO CMP results that differ significantly.

Oracle-Based Power Gating To understand the limit of energy savings from power gating cores during memory stalls, the oracle memory predictor assumes *a priori* knowledge of all memory accesses and determines the optimal power-gating behavior. The EV6 oracle policy achieves a maximum of 23.9% energy savings, and 3.6% energy savings on average. A few benchmarks show negative energy savings as high as -0.2%. These negative energy savings are caused by the lack of power-gating opportunities and the retention cells' power overhead on cpu-bound benchmarks.

The IO oracle policy achieves up to 36.8% energy savings, with an average energy savings of 8.1%. The greater energy savings result solely from the lack of memory level parallelism of the IO core, and the greater time spent in stalls waiting for the memory subsystem. Two benchmarks suffer negative energy savings as high as -0.2% for similar reasons as the EV6 core.

TAP In comparison with the Oracle, TAP must determine memory latencies in a running system to ensure that sufficient time is available to power gate a core. TAP EV6 is able to achieve 22.4% (23.9% is Oracle) maximum energy savings, and 3.1% on average. TAP does not achieve the same energy savings as the Oracle because TAP is not able to power gate memory accesses until they miss in the L3 cache, and because lower-bound latencies are used. The result is that TAP avoids any performance hit but misses out on power gating at the beginning of the core stall. TAP also sees a few benchmarks with

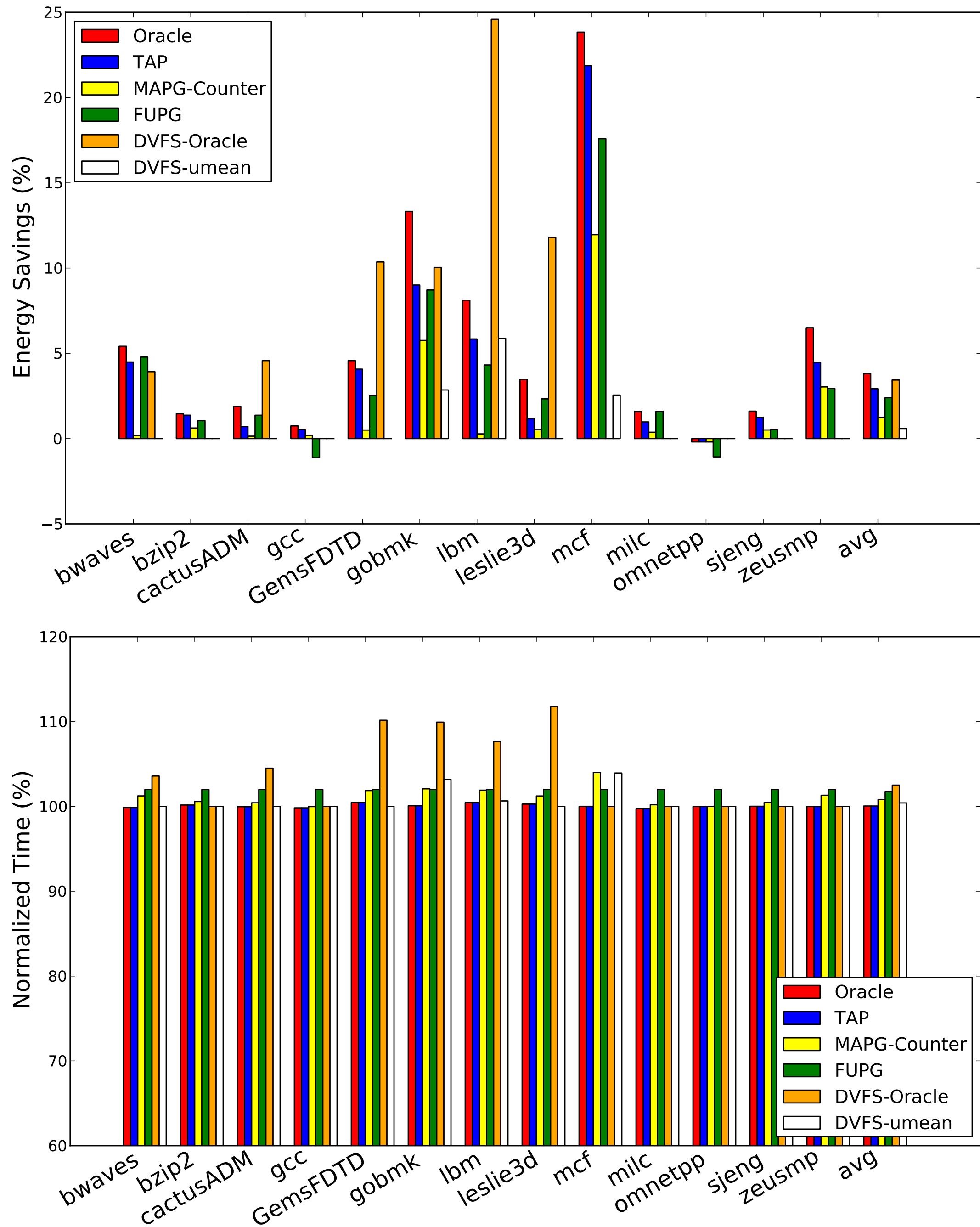


Figure 2.13. Energy savings and performance overhead of power-gating Oracle, TAP, MAPG-Counter, FUPG, DVFS-Oracle and DVFS-μmean for an EV6 CMP. Benchmarks with less than 1% absolute change are filtered out for readability.

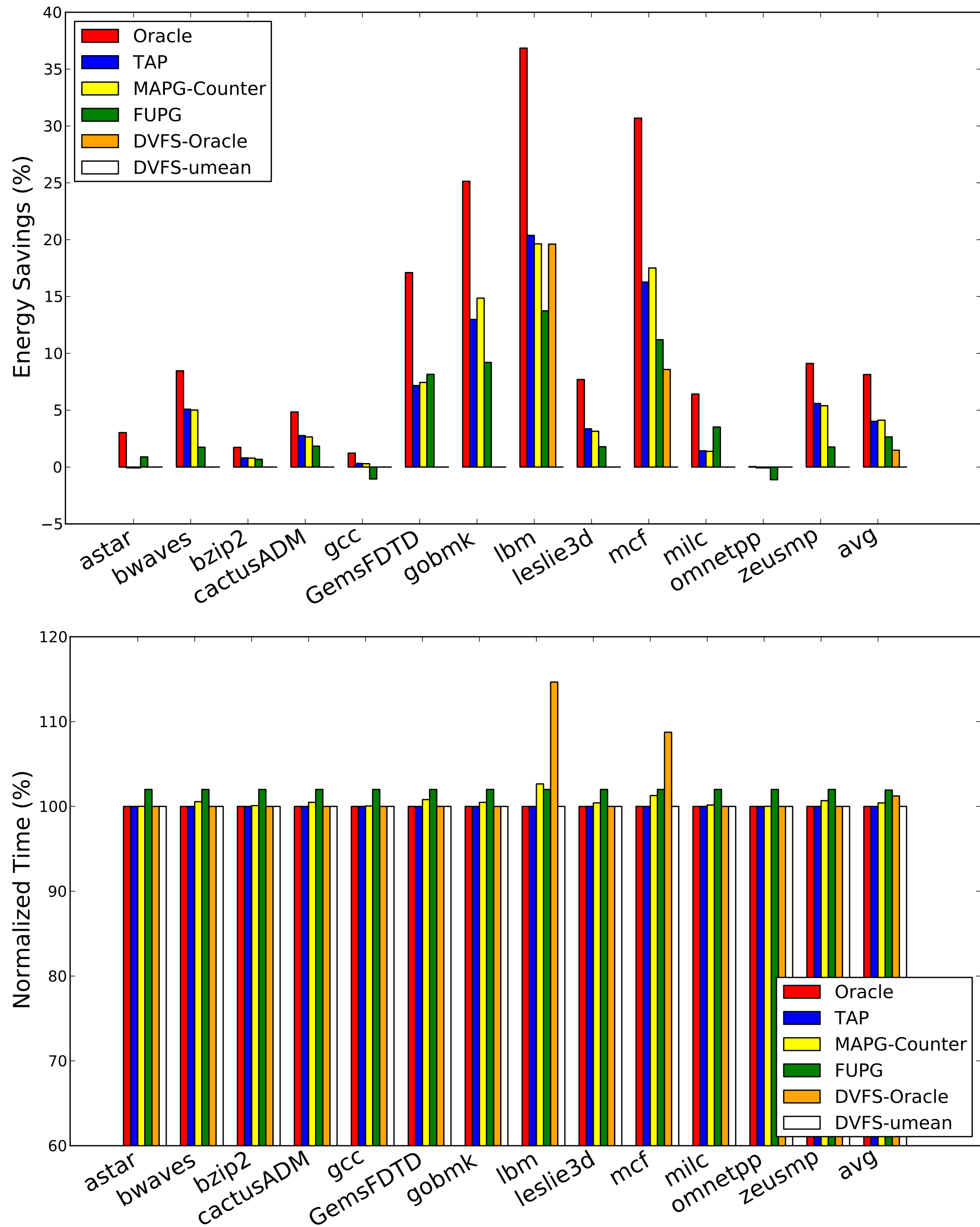


Figure 2.14. Energy savings and performance overhead of power-gating Oracle, TAP, MAPG-Counter, FUPG, DVFS-Oracle and DVFS-μmean for an IO CMP. Benchmarks with less than 1% absolute change are filtered out for readability.

-0.2% energy savings due to cpu-bound behavior.

TAP IO achieves up to 20.3% and averages 4.0% energy savings. The average energy savings for TAP IO is higher than TAP EV6 because of the lack of memory level parallelism in the IO core. However, TAP EV6 can see higher energy gains than TAP IO for the benchmark *mcf*. We account the higher peak savings of *mcf* to the EV6 core more quickly reaching the stall periods and spending a larger percentage of execution time waiting for the memory subsystem.

MAPG-Counter MAPG-Counter power gates cores after the core stalls for a longer time than the L3 hit latency, and power gates for the predicted stall duration according to its exponential-learning algorithm. MAPG-Counter EV6 is able to achieve up to 12.0% energy savings (1.2% savings on average). TAP is able to achieve 2.58× the average energy savings of MAPG-Counter for out-of-order cores.[9] The is because out-of-order cores stall more randomly than in-order cores, which makes the adaptive-counter mechanism unstable and more prone to misprediction.

In comparison, MAPG-Counter IO reduces energy consumption by up to 19.6% (4.1% on average). Although the maximum energy savings for *mcf* are not as high as TAP IO, the average energy savings are nearly identical. These similar energy savings occur because MAPG-Counter can predict the stall behavior of the IO core with more success. This indicates that the simple MAPG-Counter mechanism can act as a memory access power-gating controller for well-behaved architectures. However, MAPG-Counter experiences a perfomance hit as high as 2.6% on *lbm* (0.4% on average).

FUPG [14, 58, 83] FUPG EV6 has a maximum and average energy savings of 17.6% and 2.2% with a maximum performance hit of 2% (1.5% average), at which point control logic prevents future power-gating actions. The FUPG mechanism does better on average

[9] If method *A* saves 1% energy on average, and method *B* saves 3% energy on average, we say that "method *A* achieves 3.00× the average energy savings of method B".

for the out-of-order core than MAPG-Counter, but TAP achieves 1.4× the average energy savings of the FUPG mechanism. FUPG does achieve more energy savings than TAP on a few cpu-bound integer codes in which not all the functional units are being used, but the core does not go idle. Greater energy savings could result from the cooperation of FUPG and TAP.

FUPG IO saves up to 13.7% energy (2.7% on average). Both TAP and MAPG-Counter save 1.5× more energy than FUPG IO on average. The major reason for this behavior is that functional units account for 36% of IO core power compared to 61% of EV6 core power. Hence FUPG IO addresses a smaller portion of the leakage power problem for an IO core and better suits the EV6 for energy savings. FUPG IO also causes a 1.9% hit in performance on average (maximum 2.0% performance hit).

It should be noted that FUPG could be a complementary technique to TAP or MAPG-Counter, capable of reducing leakage power consumption when the core is not stalled, but being partially utilized. The power-gating hardware we propose should be capable of enacting a FUPG policy in addition to TAP/MAPG-Counter, but design would require careful consideration of the interaction of the WUC and the FUPG controller.

DVFS-Oracle We also examine DVFS-Oracle using the scaling properties from [30], and a controller that takes an arbitrary performance hit as long as energy is saved (see Section 2.4). The maximum and average EV6 core energy savings are 24.6% (lbm) and 3.3%, respectively. DVFS-Oracle on an EV6 core sees less maximum and average energy savings compared to the power-gating oracle, but greater savings when compared to TAP. However, these energy savings suffer from two shortcomings. First, DVFS-Oracle has a maximum and average performance hit of 11.8% (GemsFDTD) and 2.4%. Second, these energy savings rely on oracle knowledge.

DVFS-Oracle IO saves up to 19.6% power for *lbm*, but only 1.5% on average. The energy savings for lbm comes, because *lbm* performs many parallel memory accesses

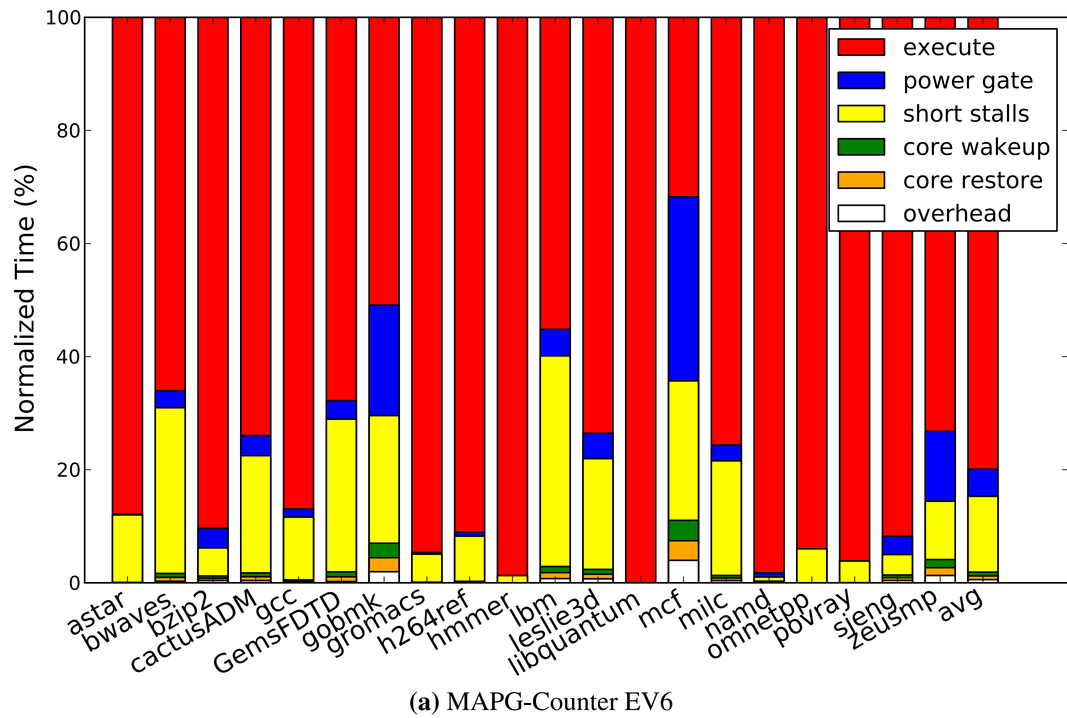
with few dependencies to simulate fluids on free surfaces. The result is that the dual issue of the IO core experiences modest memory-level parallelism and a favorable tradeoff between energy savings and performance loss. DVFS-Oracle IO also saves 8.6% energy on *mcf*, which is less than *lbm* mostly because of a greater number of dependencies in the code. The remaining benchmarks do not experience energy savings from DVFS, because the performance hit of slower clock frequencies outpaces the reduction in power. *Lbm* and *mcf* suffer from a 14.7% and 8.7% performance loss, respectively as a result of DVFS-Oracle.

DVFS-μmean [38] To understand DVFS under a realistic state-of-the-art policy, we consider DVFS-μmean [38], which predicts the performance hit of DVFS at each interval based on performance counters. DVFS-μmean achieves a maximum and average energy savings of 5.9% and 0.6% respectively. Thus, DVFS-μmean achieves less energy savings than TAP while experiencing a 1.0% performance hit on average (3.9% maximum). This result highlights the challenge of applying DVFS at 32*nm* to match different application behaviors, and why TAP's determinism can result in higher energy savings.

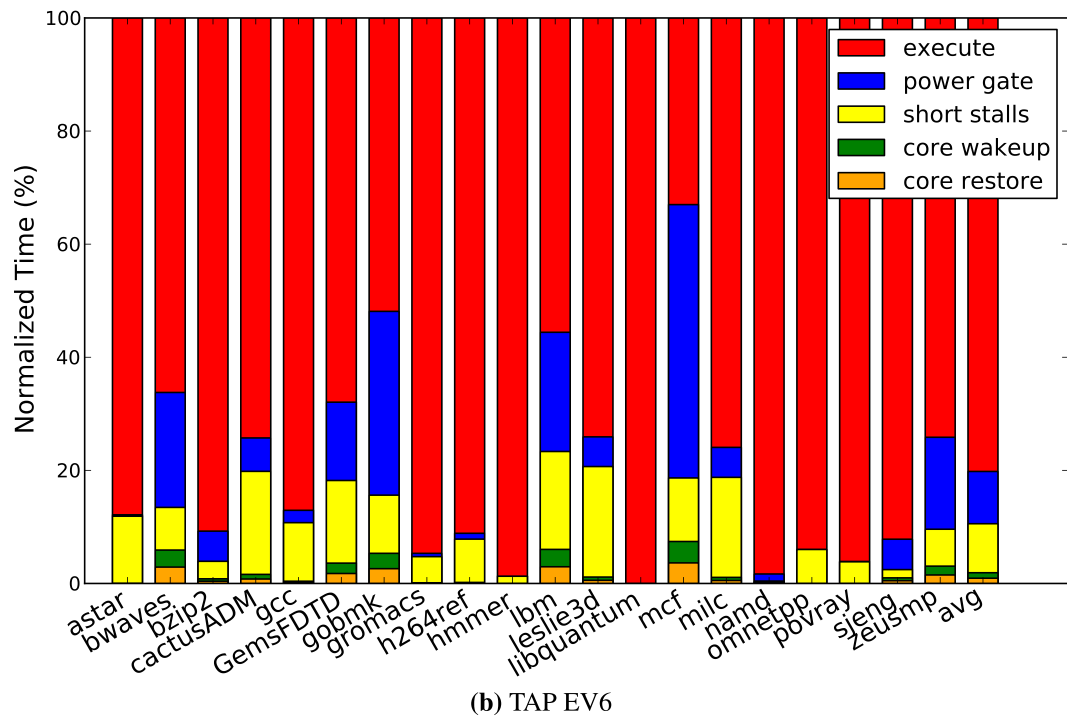
The DVFS-μmean controller decides to not use DVFS for the IO core. The reason is because DVFS-μmean depends on classifying benchmarks into different memory boundness categories, and scaling only those benchmarks that are most memory bound. However, the lack of memory-level parallelism of the IO architecture means that all benchmarks appear sensitive to core clock frequency.

2.5.2 Execution Time and Overheads

Figure 2.15 examines simulation time of each benchmark on an EV6 core and separates it into time spent executing (execute), time spent power gating the core (power gate), short stalls that could not be power gated without energy loss (short stalls), core wake-up time to charge core logic (core wakeup), core restore time to restore data from



(a) MAPG-Counter EV6



(b) TAP EV6

Figure 2.15. Breakdown of simulation time for each benchmark running on an EV6 core utilizing either MAPG-Counter or TAP to save energy.

retentive flip-flops and fill the pipeline (core restore), and execution overhead from waking up the core too late (overhead).

We observe that TAP has no measurable performance overhead for EV6 cores. The reason for this is that TAP wakes up the power-gated core for the lower bound access latency of a next-level hit in the memory hierarchy. The result is that the power-gated core is resumed in advance and always ready for the memory response. MAPG-Counter does incur a small performance overhead of up to 3.98% (0.56% on average) for the EV6 core. TAP and MAPG-Counter power gate cores up to 48.34% and 32.55% of the time for the benchmark *mcf*. Averaged across all benchmarks, they power gate cores for 9.24% and 4.82% of time, respectively. TAP is able to achieve an average improvement of 1.92× over MAPG-Counter for average time spent power gating the core.

TAP's advantage in power-gated time has a secondary effect, which is that TAP spent more time waking up and restoring the core. For the out-of-order core, TAP spends an average of 0.97% and 0.93% waking up and restoring its cores compared to MAPG-Counter's 0.68% and 0.66%. It is true that TAP only power gates the EV6 core slightly more often than MAPG-Counter. However, TAP's greater precision in tracking memory requests and expected response latency allows it to avoid harmful power-gating decisions. In comparison, MAPG-Counter has a counter that starts when the core stalls, which is subject to the variance in stall time from the EV6 core re-ordering instructions.

2.5.3 Energy Savings as a Function of Wake-up Latency

Both MAPG-Counter and TAP should save more energy as wake-up latency decreases. In this subsection, we examine both systems' sensitivities to wake-up latency, and further consider outcomes if wake-up latencies are reduced below our calculated limits. Figure 2.16 shows that energy savings increase linearly with reduced wake-up delay for both TAP and MAPG-Counter. Across the 12 benchmarks, TAP's average

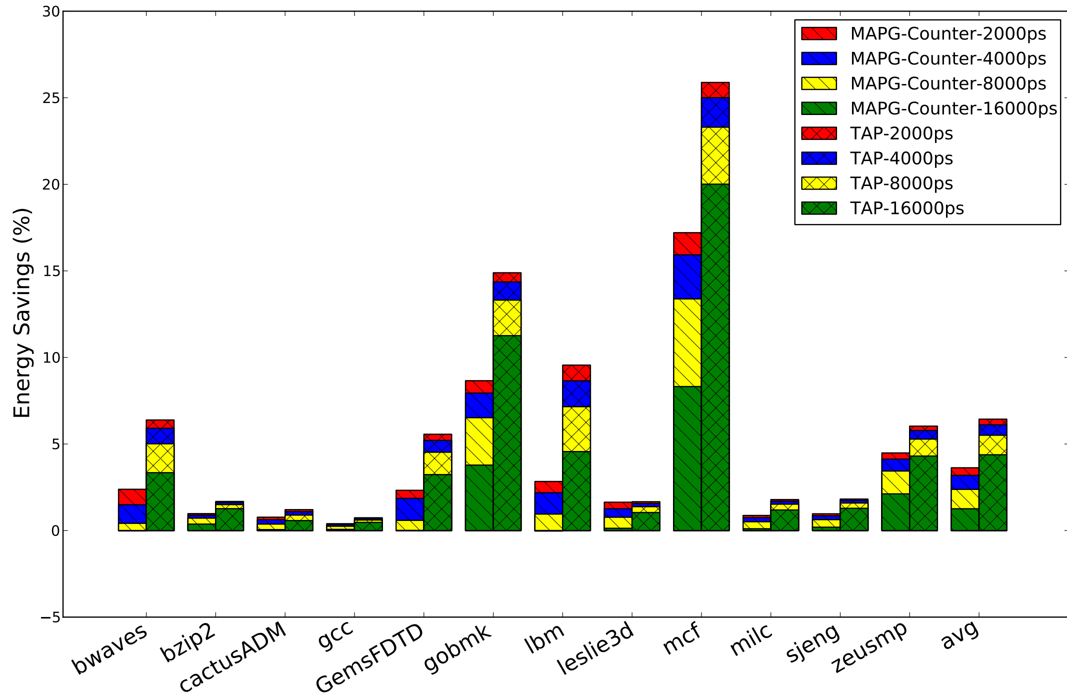


Figure 2.16. Energy savings for MAPG-Counter and TAP as wake-up mode changes from 2 *ns* to 16 *ns* wake-up latency for an EV6 core. Benchmarks *astar*, *gromacs*, *h264ref*, *hmmer*, *libquantum*, *povray*, *namd*, and *omnetpp* omitted due to small change (less than 0.2%).

energy savings increase from 2.33% to 3.33%, while MAPG-Counter's average energy savings increase from 0.70% to 2.15% between the wake-up latency of 16 *ns* and 2 *ns*. For TAP, peak energy savings for *mcf* increase from 20.00% to 25.88% as wake-up latency decreases from 16 *ns* to 2 *ns*. Likewise, MAPG-Counter's peak energy savings for *mcf* increase from 8.22% to 17.21%. In general, improved energy savings results from less wake-up time overhead, and the ability to power gate the core for longer periods of time.

A comparison between TAP and MAPG-Counter indicates that reduced wake-up latency has a more dramatic effect on the energy savings of MAPG-Counter than TAP. Indeed, as wake-up latency decreases from 16 *ns* to 2 *ns*, TAP's average energy savings increases by 1.43× on average. In contrast, MAPG-Counter's energy savings increases by 3.05×. MAPG-Counter's greater dependence on wake-up latency has two causes.

First, a lower wake-up latency reduces the penalty of overpredicting the duration of the idle interval. Second, the lower wake-up latency makes it easier to power gate for at least the break-even time so that negative energy saving periods are minimized. In comparison, TAP's greater fidelity in tracking memory requests allows it to determine that longer idle periods exist and to power gate sooner.

2.5.4 Adapting to Memory Contention

Both TAP and MAPG-Counter can adapt to varying levels of memory contention and can power gate cores for longer as memory subsystems become oversubscribed. We show how both TAP and MAPG-Counter adapt to a system facing increased memory contention for the multi-threaded memory benchmark, *STREAM*, running on a CMP with up to 32 cores. *STREAM* is a memory-intense benchmark used to measure sustained memory bandwidth and computation rates for simple vector kernels [84]. We modify *STREAM* to act as an embarrassingly parallel memory benchmark such that more threads cause more simultaneous requests to the memory subsystem. Increasing *STREAM's* thread count causes more queuing of memory requests, longer delay per request and more frequent stalling of each core. A good power-gating technique should be able to power gate the core more often to reduce power consumption.

Figure 2.17 depicts both average duration of core stalls and the percentage of total simulation time spent power gating the core for both TAP and MAPG-Counter. The x-axis tracks the number of threads that are run simultaneously. The left y-axis indicates the average stall duration for a core in nanoseconds as the number of threads increases from 0 to 32. The right y-axis shows the percentage of time that both TAP and MAPG-Counter can power gate the core as the memory subsystem experiences more contention.

First, we note that as the number of threads increases, the average duration of

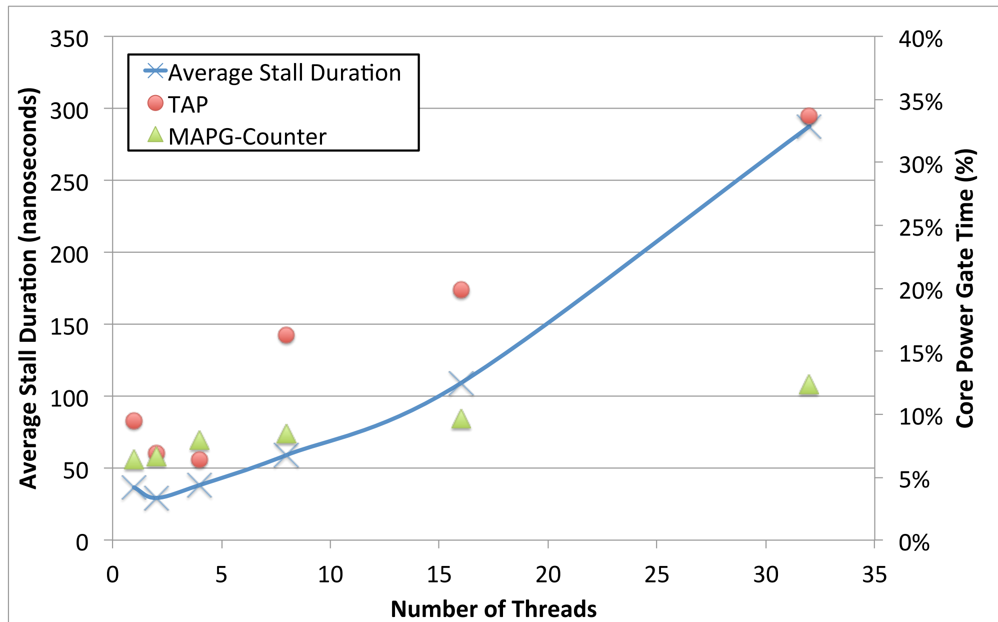


Figure 2.17. TAP and MAPG-Counter adapting to increasing memory contention. The left y-axis shows the average duration of a stall in nanoseconds, while the right y-axis shows the percentage of time that TAP and MAPG-Counter can power gate as a function of the number of *STREAM* threads.

a core stall increases. For 1, 2, 4, 8, 16, and 32 threads, the average stall durations are 36.77 *ns*, 29.31 *ns*, 38.29 *ns*, 59.191 *ns*, 109.22 *ns*, and 287.63 *ns*, respectively. From 1 to 32 threads, average core-stall duration increases by 7.82×[10]. The increase in the average core-stall duration is caused by more threads making parallel requests to the memory subsystem at once. This increase causes longer queues in the memory controller and contention to use the limited number of memory channels to transfer the requested cache line. Further, an increase in memory demand decreases the probability of a row-buffer hit and yields longer access latencies.

In addition, Figure 2.17 shows that as cores experience increased memory latency, both TAP and MAPG-Counter power gate the core longer. TAP power gates the core for 9.08%, 6.61%, 6.21%, 15.81%, 19.55%, and 33.50% of the time for 1, 2, 3, 4, 8,

[10]From 1 to 2 threads, core-stall duration decreases. This happens because more threads both increase the amount of available cache (more cores) while increasing the number of simultaneous memory requests.

16, and 32 threads, respectively. From 1 to 32 threads, TAP power gates cores 3.69× longer. MAPG-Counter power gates the core for 6.03%, 6.19%, 7.49%, 8.11%, 9.38%, and 12.18% of the time as the number of threads increases from 1 to 32, a total increase of 2.02×. Together, this data indicates that TAP adapts more closely to the level of memory contention and shows greater energy proportionality over a greater range of memory contention. However, TAP power gates its core less for 4 threads than for 2 even though average core-stall time increases. This is because TAP uses a conservative lower-bound estimate of memory-response time and does not account for all memory scheduling actions, and memory contention scenarios.

2.5.5 Distributed, Staggered Wake Up

The previous section shows that TAP can adapt to increasing memory contention, but it is not clear if the centralized WUC would be practical for such a large multicore processor. For a 16-core system with multiple cores waking up simultaneously, voltage noise on the power distribution network can cause unsafe voltage drops on neighboring active cores. By introducing a sub-nanosecond stagger between any two adjacent cores waking up, worst-case inrush current and resulting voltage noise are reduced. The result is a faster wake-up mode and increased energy savings on the chip. Hence cores should wake up staggered by at least a fraction of a nanosecond.

Figure 2.18 shows SPICE simulation of the effect of stagger on core wake-up latency for no-stagger, 0.3*ns*-stagger, 0.6*ns*-stagger, and 0.9*ns*-stagger for CMPs composed of 16 EV6 cores as 0 to 14 cores are idle. Staggered wake up can reduce the variance between the min and max wake-up latency when most cores are actively executing or waking-up. For the 16-core CMP with no cores idle and no stagger, the max and min wake-up latency is 18.4 *ns* and 9.7 *ns*, whereas, a staggered wake up of 0.9 *ns* decreases the max and min values to 10.7 *ns* and 8.6 *ns* respectively. As more cores

go idle, staggered wake up has less impact on reducing the variance of wake-up latency because cores are less likely to interfere with each other, and the core's location becomes the dominant factor in wake-up latency. For example, when 14 cores are idle in a 16-core CMP, the min and max wake-up latencies are both 4.5 *ns* and 9.1 *ns*, respectively, in both the no-stagger and 0.9*ns*-stagger cases. Lastly, stagger reduces the maximum wake-up latency as more cores are active. An example of this is for the 16-core case when no core is idle; maximum wake-up latency is 18.4 *ns* without stagger and 10.7 *ns* with 0.9 *ns* stagger, a reduction of 58.2%. Thus, stagger relaxes the guardband on wake-up latency.

Figure 2.19 shows the energy impact of staggered wake up for a 16 EV6 core, TAP architecture. The largest improvement in energy savings is 3.14% for *mcf* as energy savings increase from 18.92% to 22.06% for staggered wake ups of 0.0 *ns* and 0.9 *ns* respectively. On average, energy savings go from 3.52% to 4.34% as stagger increases from 0 *ns* to 0.9 *ns*. Staggered wake up does not cause any decrease in energy savings. Although cores may have to wake up at slightly different times, the savings in latency with which they wake up is much greater than the stagger offset of 0.9 *ns*.

To have cores safely and reliably use staggered wake up, a controller or scheme is required to control the wake-up behavior of all cores. A centralized WUC would be able to use more aggressive wake-up modes and power gate the core for longer, but such a design does not scale to many cores. By contrast, the distributed wake-up control of Section 2.3 can scale to many-core designs, but sacrifices some power-gating time waiting for an assigned wake-up slot. Figure 2.20 shows the difference of application power-gated time for a subset of the Spec2006 benchmarks in a 16-core TAP architecture. Our simulations show that the distributed wake-up control has a maximum decrease in power-gated time of 2.68% (0.93% on average) for the benchmark *lbm*. This indicates that the difference in energy savings between a distributed scheme and a centralized WUC would be negligible, but that a distributed scheme scales to many-core architectures.

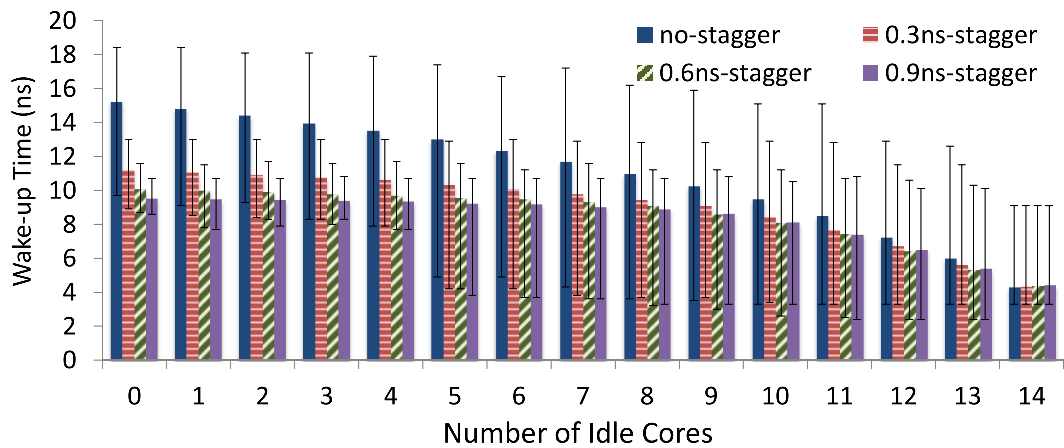


Figure 2.18. The improvement in core wake-up latency with increased stagger for a 16 EV6 core CMP as 0 to 14 cores are idle. The average latency is shown with bars denoting the min and max safe wake-up modes.

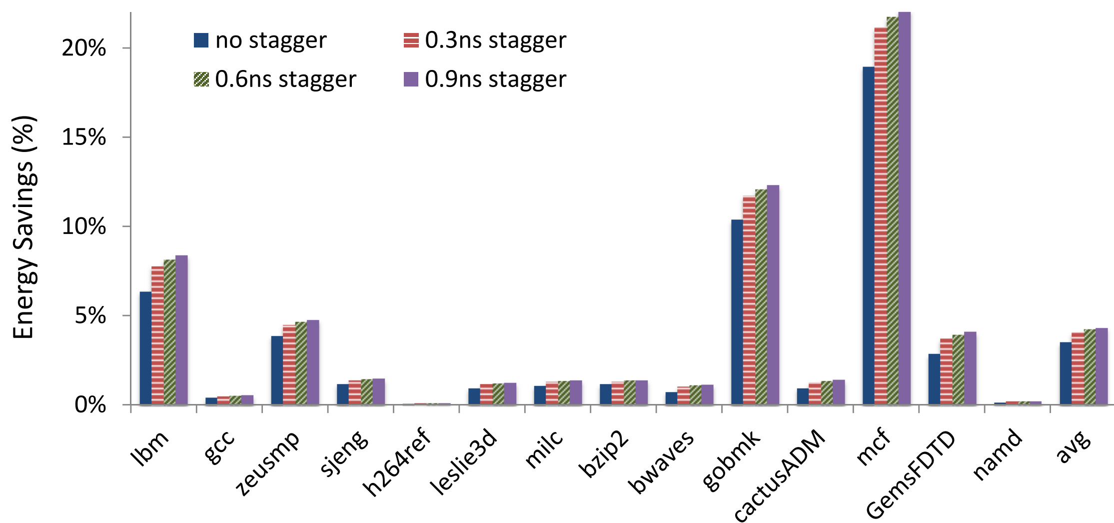


Figure 2.19. The improvement in energy savings with staggered wake ups in an 16 EV6 core CMP. Benchmarks with less than 0.2% energy savings filtered.

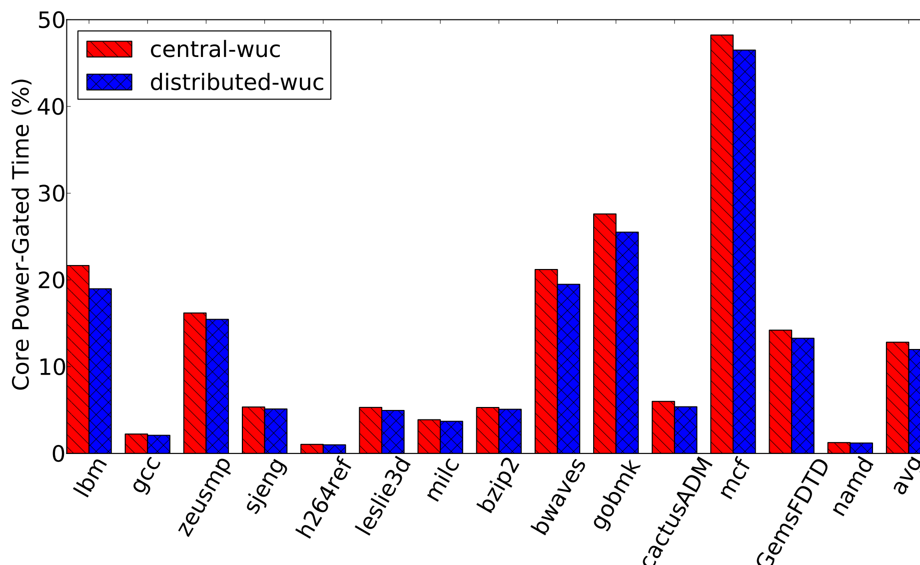


Figure 2.20. Comparison of power-gated time using staggered wake up with a centralized WUC versus a distributed WUC.

2.6 Summary

With each generation of microprocessors produced, leakage power becomes a larger issue. Data center applications and the servers that run them will not be energy efficient if leakage power is wasted, while no interesting work progresses. In this chapter, we have described two techniques, TAP and MAPG-Counter, which effectively reduce wasted leakage power for cores waiting on the memory subsystem, while actively executing workloads. MAPG-Counter achieves higher average energy savings for in-order cores (4.1%) than TAP. However, TAP achieves 22.4% maximum energy savings for an out-of-order core. The energy savings for the out-of-order core are noteworthy for both being within 13.9% of the Oracle scheme on average, and having 2.58× the average energy savings of MAPG-Counter. TAP is also shown to be more adaptive to different levels of memory contention, and yields a more energy proportional system than MAPG-Counter. Lastly, we demonstrate that a wake-up stagger of 0.9 *ns* reduces

core wake-up latency by up to 58.2% (7.7 *ns*), and increases TAP's energy savings by an additional 3.14%.

2.7 Acknowledgments

Chapter 2 contains material from "MAPG: Memory Access Power Gating", by Kwangok Jeong, Andrew B. Kahng, Seokhyeong Kang, Tajana S. Rosing, and Richard Strong, which appears in *Design, Automation & Test in Europe Conference & Exhibition*, 2012. The dissertation author was a principle contributor and author of this paper.

Chapter 2 also contains material from "TAP: Token-Based Adaptive Power Gating", by Andrew B. Kahng, Seokhyeong Kang, Tajana S. Rosing, and Richard Strong, which appears in the *ACM/IEEE International Symposium on Low Power Electronics and Design*, 2012. The dissertation author was a principle contributor and author of this paper. This material is copyright ©2012 by the Association for Computing Machinery, Inc. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Publications Dept., ACM, Inc., fax +1 (212) 869-0481, or permissions@acm.org.

In addition, Chapter 2 contains material from "Many-Core Token-Based Adaptive Power Gating", by Andrew B. Kahng, Seokhyeong Kang, Tajana S. Rosing, and Richard Strong, which will appear in in the *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. The dissertation author was a principle contributor and author of this paper.

Last, Chapter 2's research was supported by the MARCO FCRP (GSRC and MuSyC centers), Qualcomm, Oracle, and NSF grants SHF-0916127, SHF-1218666, SHF-1116667 and CCF- 1162085.

Chapter 3

Very Fast Core Switching

The previous chapter discusses a low-latency, power-gating technique that can reduce core leakage power during a memory access. Such a technique is best suited for cores running memory bound applications, and requires modification to existing processor microarchitecture. If a technique could leverage the millions of underutilized cores in today's data centers to improve energy efficiency, the benefits may be realized more quickly. Such a technique can use low-latency software to improve the energy efficiency of applications, while maintaing critical QoS agreements in the data center.

To this end, Chapter 3 presents fast core switching, a software-based thread migration technique, which enables a variety of potential improvements such as: thermal management [28, 52], fine-grained load-balancing [18], and exploiting asymmetric cores [72] by moving computation when execution characteristics change. For example, a thermal management technique may migrate the most power-hungry applications to the edge of a chip to reduce overall chip temperature, and reduce the energy consumption from cooling costs [35]. Alternatively, when the server experiences high demand for computation, thread migration every 20–200 μs can spread computation across the chip, reduce chip temperature by 12.4° C, and allow each core to run at a faster frequency [52]. In all of these cases, the migration mechanism is used to improve server energy efficiency either directly or indirectly, and its delay ultimately determines when and how often

the mechanism can be used. Therefore, successful exploitation of these opportunities demands low core-switching costs.

CPU vendors currently offer as many as 64 cores per 1U server [8]. Inter-core communication costs have dropped by orders of magnitude compared to traditional multiprocessors. This creates the potential for significantly more nimble and dynamic management of executing threads, since it reduces the time for hardware to move a migrating thread's working set by similar orders of magnitude.

However, we cannot exploit these opportunities while the software costs of moving a thread remain as high as they are now, since these costs can dominate the communication costs of moving the working set. For instance, both simulation and hardware experiments show that thread migration between complex cores takes longer than 2 μs on average. Simulations show that migration across asymmetric cores takes more than 8 μs on average. In comparison, a remote cache line for the complex core architecture of Chapter 2 is accessible in 45 *ns*, which is 44$\times$ shorter than a thread migration. This research seeks to understand and reduce these software costs to provide a lower overhead migration mechanism.

This chapter focuses particularly on the potential of asymmetric multicore processors, because core-to-core performance asymmetry can improve energy and area efficiency. An asymmetric multicore CPU has cores, which vary greatly in complexity and energy consumption, while executing more or less the same instruction set. Code that has no need for a complex core can be run on a simpler core, often with relatively little performance cost but with greater throughput per watt. The use of asymmetric multicore processors increases the need for frequent migration of threads between cores.

Previous work shows the potential value of multicore performance asymmetry, starting with Kumar *et al.* [72], who proposes an asymmetric single-ISA (ASISA) multicore. Their simulations show that dynamic migration of application code between

cores of varying complexity could improve energy efficiency. Other publications propose asymmetric architectures with specialized cores for operating system (OS) code and for network code [94, 117]. In addition, Nellans *et al.* suggest that adding OS-specific caches to standard cores might also be a good way to improve efficiency [95].

Balakrishnan *et al.* [15] point out that an asymmetric multiprocessor can outperform a symmetric processor, when a faster core executes serial portions of a parallelized application. Grant *et al.* [48] evaluate an ASISA approach in which specific kernel threads are bound to cores optimized for OS performance, using actual hardware and throttling the clock of one core to emulate the performance of a low-power core. Previous work [89] also evaluates the use of simpler cores to run OS code, using simulation to show the potential for improved energy efficiency.

Much of the prior work dynamically matches the characteristics of code to the appropriate core. Exploiting performance asymmetry requires switching cores during the execution of a thread. For example, to get good performance, the OS scheduler may need to migrate a thread from a slow core to a fast, idle core [15]. Thread migration is also necessary when balancing loads between cores in a symmetric or asymmetric system. However, dynamic migration must balance the benefits against the costs, especially if the benefits come from frequent migrations to exploit the performance characteristics of relatively short execution segments.

Therefore, core-switching costs are fundamental to evaluating the feasibility of asymmetric multicore systems. It might seem that moving thread T from core A to core B simply requires transferring the thread's architectural state from A to B. However, this would only work if core B is idle before the core switch, and if no other thread should run on core A while T is bound or blocked at core B. Otherwise, T must be the most appropriate thread to run on B, and the scheduler must find another thread, perhaps the idle thread, to run on A. These are both scheduling decisions, requiring access and

updates to the kernel's scheduling data structures. This is why the scheduler must be involved in a core switch.

In all versions of this chapter's modifications to the Linux scheduler, kernel code that wants to initiate a core switch calls a *SwitchCores* function that, sooner or later ends up invoking Linux's *schedule* function. This approach does not replace the operating system's normal scheduling policies. All standard scheduling proceeds as usual. The kernel modifications do not core switch at such points. No previous work has carefully evaluated the software costs of core switching, or how these might be reduced. This chapter makes the following contributions:

- We reduce Linux core-switching latencies by up to 2.5× compared to vanilla Linux, and see migration latencies as low as 933 *ns*.
- We show several ways to reduce the cost of software-based core switching, which indicates that extra context switches, unnecessary scheduler operations, and the interrupt code path all hinder fast thread migration.
- Using simulations, we explore how core-switching costs depend on several architectural considerations. They indicate that core switching is senstive to L1 cache sizes, the wake up latency of cores, and the presence of an instruction that allows efficient cross-core wake ups.
- We present both microbenchmarks and macrobenchmarks evaluating the efficiency of core switching, on both real and simulated hardware. Real hardware running macrobenchmarks has no performance penalty from the proposed core-switching technique. Simulations show that core switching between asymmetric cores also improves energy efficiency by 3.37× on average.

3.1 Related Work

Chapter 3 relates to research in two areas: thread migration techniques and scheduling for asymmetric multicores. Thread migration techniques focus on efficiently moving threads between cores with the goal of minimizing performance impact, or to exploit idle core resources by fine-grained thread movement. Scheduling for asymmetric multicores focuses on the challenge of scheduling the use of different complexity cores to optimize the performance and energy efficiency of an application or workload. The subsections that follow discuss each in their turn.

3.1.1 Thread Migration Techniques

There is a rich history in the literature of systems that supports thread migration on conventional multiprocessors [109]. However, the issues and priorities shift significantly on a chip multiprocessor, where extremely low hardware communication costs apply much more pressure on the software overhead of the migration mechanism.

Constantinou *et al.* [33] consider a variety of costs associated with moving threads between cores on a CMP, but focus primarily on moving and warming up caches and branch predictor state. They do not address the software costs of migration.

Li *et al.* [79] modify the Linux scheduler to create a custom scheduler, with a load balancer that accounts for the asymmetry of an ASISA system. They account for the expected costs of moving a thread, particularly the cost of moving the cache working set. Their best scheduler migrates threads many orders of magnitude more often than the original Linux scheduler.

Choi *et al.* [31] examine the specific case of migrating branch predictor state when a thread switches cores, but do not address software overhead issues. They find that setting the global history register of a migrated thread to the initial value of thread's

program counter can improve IPC by 13%.

Carbon proposes a hardware scheduler capable of scheduling parallel applications [75]. The application is broken down into a series of tasks, many of which can execute in parallel. The hardware scheduler then controls how these tasks migrate and execute across a multicore architecture.

Brown *et al.* [23] take a different approach and propose a shared-thread multiprocessor (STMP), where the hardware manages thread movement. Thread state is represented in hardware that is shared among all cores on a chip, so the hardware can move a thread between cores without direct OS involvement. For example, STMP would allow core-switching in a user-mode threads package. Later, Brown *et al.* improve fast thread migration by predicting and migrating a thread's working set [22]. They find that migrating instruction cache entries that are likely to be accessed by the migrated thread significantly boosts performance.

STMP is an intriguing alternative to software-managed core-switching, but we do not yet have the ability to experiment with how this approach interacts with OS code. While the STMP hardware can reschedule active threads whose state is represented in hardware, the OS will have to be involved with scheduling threads in several cases; such as, when threads are blocked on OS-mediated I/O operations, when there are too many threads for the hardware to represent, or when the scheduling policies must involve OS-managed state. STMP will therefore require fairly significant modifications to the operating system's view of the threads it manages.

3.1.2 Scheduling for Heterogeneous Multicores

Hill *et al.* point out that asymmetric multicores are an inevitable consequence of scaling to large numbers of cores on a chip [54]: by analogy to Amdahl's Law, highly-parallelizable applications have sequential code that runs best on a core with the

greatest possible single-thread performance. Such asymmetry exists on both Intel and AMD processors in which one or more cores can increase their frequency for critical applications.

Kumar *et al.* [72] argue for asymmetric cores because some code gains little benefit from complex cores, and if executed on a simple core, will run at nearly the same speed with lower energy consumption. Simple cores can be a better match for memory bound application code. To utilize asymmetric hardware, they present scheduling policies for ASISA architectures in both the performance [74] and power/performance [72] domains. Those scheduling policies are sampling-based, and run at infrequent intervals to limit migration costs. Their assumption is that migration points are unknown to the software for an application that rapidly changes in behavior, which is not the case for the architecture that this chapter assumes. Kumar *et al.* [72] model core switching at a very coarse granularity, allowing them to neglect the cost of core switching. However, the introduction of specialized cores for OS and similar code implies the potential for much more frequent migration, potentially making performance highly sensitive to that cost.

Becchi *et al.* [18] model core switching for a similar architecture, using a somewhat more sophisticated model including the execution of an OS. While they model the data-communication costs of core switching, they do not try to model the kernel software overheads.

Work in [89] observes that operating system code has large working sets and many branch dependencies, and hence should be a good match for energy-efficient execution on simple cores in asymmetric multicores. The authors migrate OS code from a complex core to a simple core at system calls, to improve energy efficiency. However, no attempt is made to reduce the overhead of migration between cores.

Fedorova *et al.* [45], similar to [79], create a custom OS scheduler to account for an asymmetric system. In particular, they seek more balanced core assignments to

Table 3.1. Summary of core-switching versions

Version number	**Mechanism(s) used**	Section
V1	Linux's existing thread-migration mechanism	3.2.2
V2	Direct invocation of modified scheduler	3.2.3
V3	Scheduler fast paths for source and target	3.2.4
V4	Idle loop uses polling instead of interrupts	3.2.5
V5	Cross-core wake up from quiesce	3.2.6

increase fairness and decrease runtime jitter.

Chakraborty *et al.* [27] also look at migrating OS code to distinct processors. However, their motivation is not heterogeneous cores, but the opportunity to spread computation that is unlikely to use the same working set (e.g., user and kernel code) onto separate cores with separate caches.

To the best of our knowledge, no previous work carefully evaluates the software costs of core switching, especially in the context of the operating system. In contrast, this chapter uses a trace driven technique to break down the costs of migrating a thread via the operating system. This analysis is further used to reduce core-switching costs, and minimize performance. Last, this chapter analyzes the new core-switching technique for both simulated and actual hardware.

3.2 Software Approaches to Core Switching

First, this section describes how to modify the code for various specific system calls to invoke core switching, so as to avoid modifications to user applications. Next, this section describes a sequence of increasingly more efficient designs labeled V1 to V5 (see Table 3.1). V1 starts with the current state-of-the-art Linux migration code. Each step identifies inefficiencies in the migration process, and optimizes around them. In addition, the separation of the various steps allows the reader to gauge the significance of each change. This chapter makes several assumptions related to core switching:

- Threads only switch between cores with a single ISA. That is, any core where a thread can run will correctly execute its instructions, allowing the experiments to run unmodified applications.[1]

- This chapter considers only the performance consequences of switching a thread between cores on a single die. The kernel modifications in this section can successfully switch threads between cores on different sockets, but since inter-socket communication costs are much higher, this is less useful in cases where rapid core switching is desirable.

We implement all changes as modifications to the Linux 2.6.18 kernel. The 2.6.18 kernel uses a scheduler optimized for efficiency [9]. Starting with 2.6.23, Linux uses the *Completely Fair Scheduler*, whose design the author describes as “quite radical” [90]. This chapter does not analyze this design to know how it can be optimized for efficient core switching, though we suspect its modularity might add some latency. There is no obvious reason why this section’s approach would not generalize to other operating systems.

3.2.1 Modified System Calls

Given a core-switching mechanism that is reasonably efficient but not free, it is necessary to choose when to switch. One approach is to provide a simple system call like *coreswitch(destcoreset, flags)*, to allow an application to explicitly initiate switching to one of a set of cores, with policies such as core affinity controlled by flags. This requires application-specific changes, so this dissertation chooses not to evaluate this approach.

Instead, long-running and frequently-used system calls have been modified to invoke core switching. Figure 3.1 shows how to add code to the *read* system call, to

[1] It is possible to trap on certain unimplemented instructions, save the thread state, and resume the thread on a more appropriate core; see Li *et al.* [80] for an example. It is also possible to migrate applications between different instruction set architectures [36].

```
sys_read(unsigned int fd, char __user * buf, size_t count) {
        struct file *file; ssize_t ret = -EBADF;
        int fput_needed; int switched = 0;

/*new*/ if ((count >= RWThresh) && OKtoSwitch(__NR_read))
/*new*/    switched = SwitchToOScore();

        file = fget_light(fd, &fput_needed);
        if (file) {
                loff_t pos = file_pos_read(file);
                ret = vfs_read(file, buf, count, &pos);
                file_pos_write(file, pos);
                fput_light(file, fput_needed);
        }

/*new*/ if (switched)
/*new*/   switched = SwitchToAppCore();

        return ret;
}
```

Figure 3.1. Example of a system call modified to support core switching

conditionally switch to an OS-specific core if the buffer size exceeds a threshold. The OS sets the buffer-length threshold to 4096 bytes. The costly work (e.g. *vfs_read*) then continues on the OS core; when done, if code did switch cores, it is switched back. The following system calls are modified in the same way: *open, stat, read, write, readv, writev, select, poll, fsync, fdatasync, readfrom, sendto* and *sendfile.*

3.2.2 V1: Linux's Thread-Migration Mechanism

The first approach is to use Linux's existing thread-migration mechanism, normally used for relatively long-term load-balancing across cores. Linux's thread-migration mechanism is the current state-of-the-art for software core switching. When a task wants to migrate, it puts itself on a per-core migration queue, wakes up, and switches control to a per-core *migration thread*, which does the actual work of moving the thread to the run queue of the target core. If the target core is idle, the migration thread signals that core to invoke the scheduler (see Section 3.2.5 for details), which finds the thread on the target

run queue and reawakens it.[2] This migration approach involves an extra context switch between the initiating thread and the migration thread. Kernel code that wants to initiate a core switch calls a *SwitchCores* function that, sooner or later ends up invoking Linux's *schedule* function The new *SwitchCores* function requires less than 30 non-commented source code lines, that invokes this thread-migration mechanism. Thus, there is little performance overhead from the *SwitchCores* function, and migration cost reflects the default Linux mechanism.

3.2.3 V2: Modified Scheduler

V1 spends significant time context switching to the kernel *migration thread*. The V2 scheduler removes the extra context switch, and initiates thread migration directly. This means that *SwitchCores* directly invokes a modified version of Linux's *schedule* function.[3] The changes require 43 non-commented source-code lines.

Since it would be very difficult to change the arguments to *schedule*, a new thread-info field is created to pass the target core ID from *SwitchCores*. If this field is set, *schedule* unconditionally deactivates the thread T, places it on a special per-core *Alternate Queue* (*AQ*), and then rejoins the original scheduler code where it picks the next thread *N* to run on the source core. Once *schedule* has context switched the core to thread *N*, the modified version checks *AQ*, finds T safely dormant in the queue, inserts it in the run queue for the target core, and signals the target core (see Section 3.2.5). This cross-core signal causes *schedule* to run on the target core; it finds T on its run-queue, context switches to T, and the core switch is complete.

[2]This mechanism is also invoked when an application uses the *sched_setaffinity* system call in a way that requires it to vacate the current core.

[3]Apparently, Solaris uses this approach for thread migrations [44, 85].

3.2.4 V3: Scheduler Fast Paths

Through analysis of instruction-level traces from simulation runs (see Sections 3.3 and 3.4), it is clear that the V2 scheduler executes several sequences of slow and unnecessary code. The V3 scheduler realizes a fast-path version of *schedule* that it tailors to expediency for the specific case of core switching. The scheduler is split into three versions: the original modified V2 *schedule*; a *fast_schedule_source* version the source core calls to initiate a core switch, and a *fast_schedule_target* version called by the target core in response to the cross-core signal.

Both *fast_schedule_source* and *fast_schedule_target* omit a number of housekeeping functions normally done in *schedule*, like recalculating thread priority, which involves expensive arithmetic, and load-balancing for an about-to-be-idle core. Also, *fast_schedule_source* sets a special per-core hint for the target core, which tells the target core's idle loop to invoke *fast_schedule_target* rather than *schedule*. Since this is a hint, no locks are required, although the slow path, V2 scheduler, may be used occasionally on the target core. Standard scheduling events, e.g., the expiration of a thread's quantum, always use the normal slow-path V2 *schedule*, and hence all housekeeping functions execute approximately as often as they would normally.

The *fast_schedule_target* function cannot omit the code that checks *AQ*, because although the application thread initiates the core switch using *fast_schedule_source*, the check of *AQ* happens after the context switch, and thus in the idle thread, which calls *fast_schedule_target* before it yields the core, and thus, it is still within *fast_schedule_target*'s call stack. The function *fast_schedule_target* always executes in the context of the idle thread on one core or another. For the same reason, *fast_schedule_source* can omit the code that checks *AQ*. Figure 3.2 depicts the timeline schematically for two core switches. Core 0 executes code, starts a core switch through the *fast_schedule_source*,

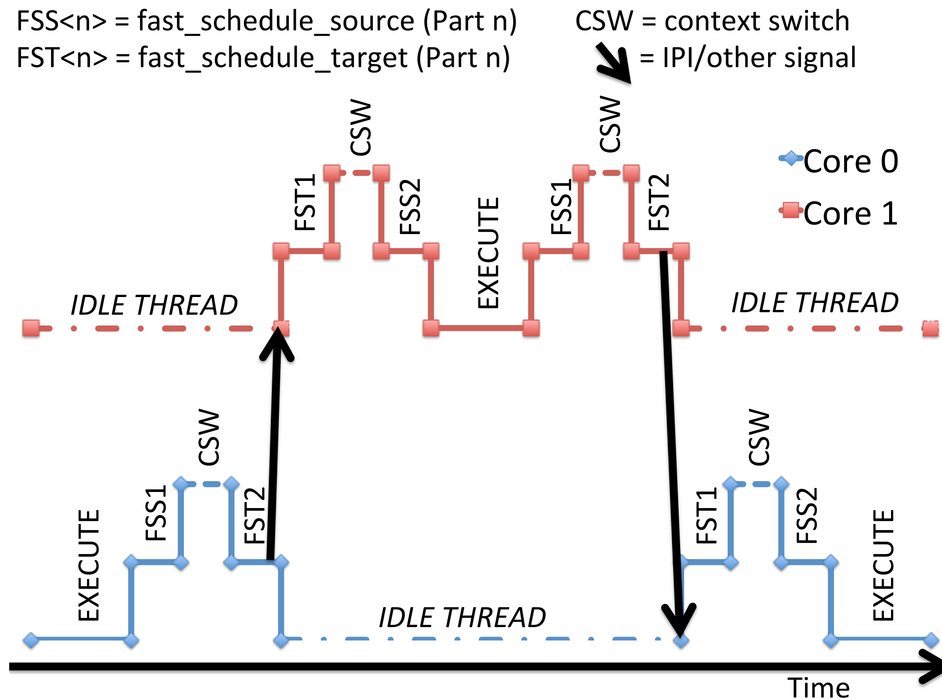


Figure 3.2. Timeline showing 2 core switches between a pair of cores, using fast-path versions of *schedule*

context switches, and *fast_schedule_target* performs the core switch. A signal wakes core 1, calls *fast_schedule_target*, and context switches the original thread for execution. The timeline goes on to show a core switch back to the original core.

Figure 3.3 (left) shows abstract pseudo-code for the V2 scheduler with modifications for core switching underlined. Figure 3.3 (right) shows the code that is deleted for *fast_schedule_source*.

3.2.5 V4: Addressing IPI Costs

The V3 scheduler needs to wake up the target core if it is currently idle. Linux allows the idle loop either to poll for changes to a *need_resched* flag, or to quiesce the core and wait for an inter-processor interrupt (IPI). Architectures that support core power-down generally quiesce idle cores. Like the scheduler, the IPI code path includes

1: void schedule(void) {
2: sanity checks
3: `prev` = current thread;
4: more sanity checks
5: compute `prev`'s run-time during quantum
6: **if** `prev` wants to switch cores **then**
7:     deactivate `prev` from `RQ[this_core]`
8:     place `prev` on `AQ[this_core]`
9: **else**
10:     handle signal-related state changes for `prev`
11: **end if**
12: **if** no other runnable threads for this core **then**
13:     try to borrow some threads from other cores
14: **end if**
15: **if** still nothing to run **then**
16:     `next = idle`
17: **else**
18:     `next` = highest-priority runnable thread for this core
19: **end if**
20: adjust priority for `next`
21: SMT-related optimizations
22: prefetch `next`'s kernel thread-related state and stack
23: prepare and do actual context switch
24: **if** `AQ[this_core]` non-empty **then**
25:     dequeue thread `t` from `AQ[this_core]`
26:     update load-balancing info for `t`
27:     place `t` on `RQ[t->target_core]`
28:     signal `target_core` to check its `RQ`
29: **end if**
30: }

Modifications for core-switching are underlined

1: void fast_schedule_source(void) {
2: ~~sanity checks~~
3: `prev` = current thread;
4: ~~more sanity checks~~
5: ~~compute `prev`'s run-time during quantum~~
6: **if** `prev` wants to switch cores **then**
7:     deactivate `prev` from `RQ[this_core]`
8:     place `prev` on `AQ[this_core]`
9: **else**
10:     ~~handle signal-related state changes for `prev`~~
11: **end if**
12: **if** ~~no other runnable threads for this core~~ **then**
13:     ~~try to borrow some threads from other cores~~
14: **end if**
15: **if** still nothing to run **then**
16:     `next = idle`
17: **else**
18:     `next` = highest-priority runnable thread for this core
19: **end if**
20: adjust priority for `next`
21: ~~SMT-related optimizations~~
22: prefetch `next`'s kernel thread-related state and stack
23: prepare and do actual context switch
24: **if** ~~`AQ[this_core]` non-empty~~ **then**
25:     ~~dequeue thread `t` from `AQ[this_core]`~~
26:     ~~update load-balancing info for `t`~~
27:     ~~place `t` on `RQ[t->target_core]`~~
28:     ~~signal `target_core` to check its `RQ`~~
29: **end if**
30: }

Source-core fast path; deletions are ~~struck out~~

Figure 3.3. Abstract pseudo-code for modified versions of *schedule()*

significant software overhead, which is unnecessary in this instance.[4]

First, the existing code to send an IPI to a specific core invokes a function that sends the IPI to all members of a specified set of cores. Although the set, in this case, is a singleton, the required bit-map manipulations are very slow. Consequently, the IPI-sending function is modified to be more efficient for the singleton case. All of the benchmark results for IPI-based core switching use this improved code.

Using the IPI to invoke *schedule* on the target core results in a long code path for interrupt handling. To avoid the IPI path completely, V4 *schedule* forces the use of polling in the simulated system. The real x86 system already uses polling, due to a lack of quiesce instruction on the particular server used. Section 3.2.6 discusses a simple hardware change that can yield the time efficiency of polling without requiring the idle core to stay powered-up, and therefore without the power-increasing drawbacks of V4's naive polling.

3.2.6 V5: Cross-Core Wake Up from Quiesce

As discussed in Section 3.2.5, Linux's idle loop either polls on a *need_resched* flag, or waits for a cross core interrupt. The former mechanism is inefficient, especially for cores that can power-down. The latter is slow, because of the interrupt-handler overhead, which lasts for 160 ns on a simulated 3 GHz Alpha.

We add a new cross-core wake-up instruction to the simulated processor architecture, *wakeup(core)*.[5] This causes the specified core to continue from a quiesce instruction; it is a no-op if that core is not quiesced. The simulator continues execution on the awakened core after a delay to account for stabilizing the core upon power up. The delay

[4] The simulator's kernel quiesces in the idle loop, since this greatly reduces the cost of simulating idle cores. That, through analysis of simulation traces, is what led to the discovery that the IPI path adds a lot of overhead.

[5] This is analogous to the Intel monitor/mwait mechanism, but it is simpler to simulate. The goal is to show how core-switching software can exploit this kind of feature.

Table 3.2. A mapping from architectural configuration name to core types for both the application and OS core

Cofiguration	App Core	OS Core
sim_C	EV6 3 GHz	N/A
sim_CC	EV6 3 GHz	EV6 3 GHz
sim_SS	EV4 3 GHz	EV4 3 GHz
sim_Cc	EV6 3 GHz	EV6 750 MHz
sim_CS	EV6 3 GHz	EV4 3 GHz
sim_Cs	EV6 3 GHz	EV4 750 MHz

mechanism is designed as described in Section 3.3.1.

We modify *fast_schedule_source* to issue a cross-core wake-up instruction as early as possible. This completely hides the target-core stabilization delay, by overlapping it with a considerable amount of instruction execution on the source core, except when the source core is much faster than the target core. When this is successful, it avoids the interrupt-handler overhead entirely. It is possible that core wake up could occur later in *fast_schedule_source* to save a little power, but it is hard to calculate the optimal point without a risk of doing this too late. The target core might start running before the *need_resched* flag is set, so a separate per-core flag tells it to temporarily poll rather than quiesce.

3.3 Simulation Environment and Workloads

Simulations use the M5 simulator [20], which supports execution of full systems, including operating system code, and can simulate detailed architectural models cycle-by-cycle. We leverage M5 to generate detailed timelines, showing when interesting events such as procedure calls, cache misses, and long-latency instructions occur; these timelines have been valuable in understanding where time is being spent. M5 can also generate detailed traces showing, for example, when specific cores are idle or active.

The simulations use a model based on the Alpha EV6 (21264) as the *complex*

core, and an EV4-based (21064) model for the *simple* core. The complex core has 64 KB, 2-way set associative, 64 B block size L1 caches, with 1-cycle access for the L1 instruction cache and 2-cycle access for the L1 data cache. The simple core has 8 KB, direct-mapped, 64 B block size, 1-cycle L1 caches, except for Section 3.5, which evaluates simple cores with larger L1 caches. The simulations assume a shared L2 cache with 3.5 MB, 7-way set associative, 4 ns access latency, and a main-memory access time of 25 ns.

This chapter simulates a variety of configurations (see Table 3.2), with a naming scheme showing the types (*C* for complex, *S* for simple) and speeds (upper-case letters for 3 GHz, lower-case for 750 MHz) of each core: *sim_C* with a single (uniprocessor) 3 GHz complex core; *sim_CC* with two 3 GHz complex cores; *sim_SS* with two 3 GHz simple cores; *sim_Cc* with two complex cores; one at 3 GHz and one at 750 MHz; *sim_CS* with one complex core and one simple core, both at 3 GHz; and *sim_Cs* with one 3 GHz complex core and one 750 MHz simple core. The first letter represents the application core; the second represents the OS core.

For each dual-processor hardware configuration, this chapter runs tests using unmodified Linux, an unmodified Linux that binds Ethernet and disk interrupts to the OS core but does not core switch, and the five versions of core switching Linux (*V1* ... *V5*), also with interrupts bound to the OS core. All configurations but unmodified Linux pin the user application process to the application core. The simulator is fully deterministic and requires only one trial per experiment.

3.3.1 Modeling Core Power Up

Chapter 1 discusses that servers will often have one or more idle cores, and so power could be saved by powering-down these idle cores. This leads to some issues in modeling the performance aspects of powering up a core. These experiments assume

that a core's L1 cache state persists during power-down, using a source biasing technique [101] similar to that used in Chapter 2.[6] The experiments also assume that architectural register state is preserved similar to either the *C6* state in the Intel Core Duo [61] or the slave-latches used in Chapter 2.

When a powered-down core is powered up, some time passes until the voltages have stabilized enough for the core to safely execute instructions as discussed in Chapter 2. This delay is imposed by the RC time constant defined by the capacitance of the core and the resistance in the power wiring. This chapter uses a delay model provided by Matteo Monchiero [91]. With parameters chosen for a 65 nm process and a maximum core current of 10 A similar to the physical x86 hardware, the model predicts a power-on time of about 16.3 ns, or about 50 cycles at 3 GHz and 12.5 cycles at 750 MHz. For comparison, James *et al.* report that "a single POWER6 core is capable of causing a 13 W power step within about 20 clock cycles" [64]. This delay is longer than that calculated in Chapter 2, because of the different technology node assumptions.

We modify M5 to simulate a configurable delay between the time that a quiesced core receives an interrupt and the time that it executes the first interrupt-handler instruction. This delay ranges from 0 ns to a conservative 1000 ns to generalize findings across a range of power distribution network designs (see Figure 2.6). Section 3.5.2 discusses how this chapter's results depend on this delay. Simulation results present elsewhere in this chapter use a delay of zero to represent the ideal scenario for the power-gating techniques of Chapter 2.

3.3.2 Workloads

This chapter uses the following OS-intensive benchmarks: *netperf/TCPstream* benchmark, which sends TCP data as fast as possible, and *netperf/TCPmaerts*, which

[6] In other work, we investigate the case where core power down causes the L1 cache to flush its state [35].

receives data as fast as possible; *Web*, Apache with a workload based on SPECweb; and *DB*, using the *ex_tpcb* “TPC-B-like” example from the Berkeley DB distribution [96]. Table 3.3 shows that these applications spend a lot of their non-idle time in kernel or interrupt modes, although at slower network interface (NIC) speeds the *netperf* benchmarks often leave the CPU idle.

Table 3.3. Fraction of CPU time spent in various modes. Measurements are based on unmodified Linux on a *simulated* uniprocessor

NIC		Percentage of CPU time spent in mode			
speed	**Benchmark**	**User**	**Kernel**	**Interrupt**	**Idle**
1 Gbps	*TCPmaerts*	14.8%	27.4%	17.5%	40.4%
1	*TCPstream*	0.1%	38.1%	18.2%	43.7%
1	*Web*	55.0%	25.8%	10.6%	8.6%
10 Gbps	*TCPmaerts*	25.0%	46.2%	12.9%	15.9%
10	*TCPstream*	0.1%	27.0%	25.4%	47.5%
10	*Web*	54.5%	29.8%	8.3%	7.4%

Simulation of TCP-based benchmarks can be problematic, as described by Hsu *et al.* [57]. TCP performance depends on, among other things, the apparent round-trip time (RTT). Changes in the RTT during a connection can cause packet losses or spurious retransmissions. Unfortunately, some of the techniques the experiments use to make simulations more efficient cause changes in apparent RTT. The simulations boot the system and start the benchmark in a fast simulation mode, then checkpoint and switch to a detailed simulation, possibly using slower CPU cores. This sudden change in CPU speed can lead to increased software delays in packet processing, causing a sudden increase in the apparent RTT.

This chapter’s experiments took some steps to avoid this problem, such as slowing down the simulated core clock speeds during the pre-checkpoint phase, and increasing the simulated LAN latency; these steps mean that the relative effects on RTT of core speed are reduced. However, increasing the total RTT too much means that TCP connections

do not ramp up fast enough, leaving the cores idle.

3.3.3 Organization of Experiments

The sections that follow show results for a microbenchmark, the effect of architectural parameters on that microbenchmark, and then macrobenchmarks. The microbenchmark results for Section 3.4 accurately measure the software overhead to perform a core switch, and how these change based on core complexity. Next, Section 3.5 demonstrates that the latency of core switching also depends on factors beyond core complexity such as the size of the L1 cache and core wake-up delay. Last, Section 3.6 shows macrobenchmark results for the performance and energy impact of core-switching on OS-intense codes.

3.4 Microbenchmark Results

Core switching is not free; it adds direct overhead due to extra instruction execution, and indirect overhead due to loss of cache affinity. It can also improve performance if the use of distinct L1 caches on source and target cores reduces the number of conflict and capacity misses. This section measures the direct overhead of core switching using a microbenchmark, which invokes core switching as rapidly as possible. Since the modified system calls listed in Section 3.2.1 are chosen because they spend a lot of time in the kernel, none of these are suitable. Instead, we modify the *gettid* (get thread ID) call, since unlike *getpid*, it is not cached by the user-mode library, and it executes very few kernel-mode instructions. We then wrote *gettidbench*, which executes *gettid* N times in a tight loop, measures the total elapsed time, and computes the mean time per call. This yields twice the mean time per core switch, since the system call switches to the OS core and then back to the application core. Note that, on asymmetric hardware, the cost of a single switch may depend on its direction.

Table 3.4. Microbenchmark results for *gettid* per-call delay with 1,000,000 samples per trial. The x86 servers did not support a quiesce instruction, so the V3 *scheduler* must use polling.

(a) Dual-core x86 hardware

Over 10 trials	Mean *gettid* delay in nsec.			
	Core-switching mechanism			
	None	**V1**	**V2**	**V3**
min.	83	4094	3355	2870
max.	87	4320	3413	2886

(b) Simulated hardware

Hardware config.	*gettid* delay in nsec.				
	Core-switching mechanism				
	None	**V1**	**V2**	**V3**	**V4**
sim_CC	120	4669	3247	2550	1986
sim_SS	130	13229	10248	7982	6770
sim_CS	121	8651	6343	4696	3869
sim_Cs	118	16735	11148	8140	6781

3.4.1 Results on Real x86 Hardware

This section's experiments run *gettidbench* ($N = 1,000,000$) on a dual-core Xeon model 5160 (3.0 GHz, 64 KB L1 caches, 4 MB L2 cache), with Linux compiled in 32-bit mode. Table 3.4(a) shows the results. Even in single-user mode, some variation exists between trials, so these results report the minimum and maximum values for the mean *gettid* delay. We believe the minimum values are more likely to provide a noise-free comparison. The V3 fast-path scheduler (Section 3.2.4) experiences a round-trip delay of 2870 ns, for an excess of 1394 ns per core switch over the unmodified call.[7] For the entire system call, this is a 1.17× speedup over mechanism V2 (Section 3.2.3) and a speedup of 1.43× over the time for mechanism V1 (Section 3.2.2).

3.4.2 Results on Simulated Hardware

This section runs *gettidbench* on various configurations of the simulated hardware. In this case, the simulator can measure the duration of one call to *gettidbench*, which avoids the noise involved in measuring the average of many trials. The measured duration comes after first warming up the caches with 100 calls. Table 3.4(b) shows the results.

The delays for the *sim_CC* configuration are quite similar to those for the real

[7]Remember that each *gettid* call in this benchmark results in *two* core switches.

Table 3.5. Microbenchmark results with cross-core wake up

Hardware config.	*gettid* delay in nsec. Core-switching mechanism		
	V3	**V4**	**V5 (wake up)**
sim_CC	2550	1986	2042
sim_SS	7982	6770	6689
sim_CS	4696	3869	4087
sim_Cs	8140	6781	7024

hardware in Table 3.4(a); since both are 3 GHz dual-core CPUs with different instruction sets, this result helps to validate the simulations. The lowest delays are for V4, which uses polling instead of interrupts. On the *sim_CC* configuration, this represents an excess of 933 ns per core switch, while on the slowest hardware, *sim_Cs*, the excess is 3332 ns. Both of these represent substantial improvements over the existing Linux thread-migration V1 mechanism and over the V2 modified scheduler.

Table 3.5 reveals that while the V5 scheduler using cross-core wake ups is not always faster than the always-polling (V4) version, both are consistently faster than the fastest interrupt-based version (V3). While V5 generally follows the same approach as V4, it executes a few extra instructions in the critical path to both issue the cross-core wake up, and to reset some state flags after the wake up takes effect. Generally, given these similar delays, cross-core wake ups will be more energy-efficient than polling if idle cores can be powered down.

3.5 Effects of Architectural Parameters

This section looks at the effect of two architectural parameters on the performance of the simulated microbenchmarks: L1 cache size and core wake-up delay. Given that source biasing can reduce cache power consumption when not in use, it makes sense to increase the cache size of the OS core if it can benefit the core-switching technique. In

addition, since we assume that cores power gate when not in use to avoid extra energy consumption, it is necessary to examine what effect the core's wake-up latency can have on the performance of the core-switching techniques. The remainder of this section considers these issues.

3.5.1 L1 Cache Sizes

Table 3.6. Effect of L1 cache size on microbenchmark results, using the V5 core-switching mechanism

Hardware config.	*gettid* delay in nsec.		
	8 KB L1 caches	**16 KB L1 caches**	**16 KB 2-way L1 caches**
sim_SS	6689	5692	3787
sim_CS	4087	3665	2706
sim_Cs	7024	6515	5515

Migration costs are sensitive to cache size, for both instructions and data. The L1 caches (8 KB, direct-mapped, 64 B block size, 1-cycle) for the simple cores are relatively small, so we also simulate two versions of larger caches. Both are 16 KB total size, 64 B block size. One is direct-mapped, the other is two-way set associative. Table 3.6 shows that for the V5 mechanism, increasing the simple-core L1 cache size to 16 KB does indeed improve performance by 7% for *sim_Cs* and by 15% for *sim_SS*. There is no line in this table for *sim_CC*, since that configuration has only complex cores, which is always modeled with 64 KB, 2-way L1 caches. Adding associativity further improves performance by 15% for *sim_Cs* and by 33% for *sim_SS*. The large impact of associativity implies that the small cache experiences a lot of conflict misses.

Based on examination of detailed miss-rate statistics, we speculate that the 8 KB instruction cache creates lots of capacity misses, which are mostly eliminated by the 16 KB direct-mapped cache. However, the data cache miss rate benefits both from the larger

Table 3.7. Effect of power-up delay on performance

HW config.	Wake-up delay (cycles)	gettid delay in nsec. Core-switching mechanism None	V1	V2	V3	V4	V5
sim_CC	0	120	4669	3247	2550	1986	2042
sim_CC	1000	120	5216	3914	3224	1986	2059
sim_SS	0	130	13229	10248	7982	6770	6689
sim_SS	1000	130	13977	10916	8630	6770	6690
sim_CS	0	121	8651	6343	4696	3869	4087
sim_CS	1000	121	9376	6847	5393	3869	4081
sim_Cs	0	118	16735	11148	8140	6781	7024
sim_Cs	250	118	16891	11480	8555	6781	7930
sim_Cs	1000	118	16514	12835	9822	6781	9503

size and again from the associativity, implying that it suffers from the conflict misses.

3.5.2 Core Wake-Up Delay

This section runs M5 simulations in which the wake-up delay is very conservatively set to 1000 cycles for all cores (at both 3 GHz and 750 MHz). Table 3.7 shows the results. Increasing the delay has no effect on the non-switching and V4 (always-polling) versions, since neither of these ever quiesces a core. Similarly, it adds roughly the expected delay for the V1, V2, and V3 configurations; 333 ns to wake up the 3 GHz cores, and 1333 ns to wake up the 750 MHz core. When using the cross-core wake up V5, there is essentially no effect from the added wake-up delay, since the wake up is generated much earlier than necessary to cover 1000 cycle delay.

In the one case of the *sim_Cs* configuration in which one core is much faster than the other, when switching from the fast core to the slow core, the cross-core wake up does not happen soon enough to finish the 1000-cycle delay before the IPI is generated. Table 3.7 also shows a 250-cycle delay for the *sim_Cs* architecture, and found that in this case, the core does wake up before the IPI is sent, but just slightly too late to set the *polling* flag before the source core decides to send the IPI.

3.6 Macrobenchmark Results

This section presents results obtained by running a variety of macrobenchmarks on both a real dual-core x86 server and on a number of simulated configurations. These are throughput-oriented benchmarks, which represent realistic execution scenarios and are designed to stress the code that core switches during lengthy system calls. In the simulations, this means running some OS code on the simpler core; on the real hardware, both cores are equally fast.

These macrobenchmarks do not prove that *real applications* in general can profit from frequent low-latency core switching. To do so would require simulations with realistic workloads, which would be much lengthier than M5 has been able to support to date.

The core-switching configurations are not meant to give better throughput than the non-switching systems, but rather to enable more frequent power-down of complex cores. Therefore, the macrobenchmark results do not require throughput improvements from the various implementations of core switching; they only care that core switching does not significantly reduce performance.

3.6.1 Web Benchmark

This section simulates the *Web* benchmark, which is Apache with a workload loosely based on SPECweb[8], for a simulated time of 133 ms, after a warmup period of 333 ms using M5's simpler core model. Table 3.8 shows the results. The NIC speed has no significant effect, because this benchmark nearly saturates the cores, as shown

[8] Very loosely, it turns out. We discovered that all responses are "404 Not found" due to a buggy mod_specweb99.so. For complex reasons, we cannot fix this bug. It has the effect of making the benchmark more latency-sensitive, since the responses are all short, and it also makes comparisons between relatively brief simulation trials simpler, because all responses are about the same length. However, in order to trigger any core switching at all on server response transmissions, we therefore set the core-switching threshold for reads and writes to 256 B, rather than 4 KB. This setting is possibly too aggressive for efficient operation, but it ensures that this benchmark does reflect core-switching costs.

Table 3.8. Simulated Web results on dual-core CPUs for 1G and 10G NICs. Values are KB transferred during 133 ms.

Hardware config.	NIC speed = 1 GBit/sec Kernel configuration						
	Unmod.	**Bound**	**V1**	**V2**	**V3**	**V4**	**V5**
sim_C	1231	NA	NA	NA	NA	NA	NA
sim_CC	2087	1354	706	486	1300	1329	1301
sim_SS	1079	615	490	572	578	585	583
sim_Cc	1560	1393	724	780	852	870	(M5 bug)
sim_CS	1706	1340	847	1163	1184	1176	1161
sim_Cs	1417	1373	482	696	690	704	688
sim_CS (16 KB L1)	1772	1344	953	1192	1211	1204	1196
sim_Cs (16 KB L1)	1464	1382	574	734	764	762	754

Hardware config.	NIC speed = 10 GBit/sec Kernel configuration						
	Unmod.	**Bound**	**V1**	**V2**	**V3**	**V4**	**V5**
sim_C	1229	NA	NA	NA	NA	NA	NA
sim_CC	2326	1339	1046	1261	1296	1319	1117
sim_SS	1062	617	476	567	585	587	594
sim_Cc	1629	1391	579	155	701	696	938
sim_CS	1759	1339	872	1162	1171	1178	1169
sim_Cs	1481	1379	481	695	712	700	703
sim_CS (16 KB L1)	1777	1330	964	1191	1199	1204	1196
sim_Cs (16 KB L1)	1505	1375	513	764	791	774	753

in Table 3.3. Note that all of the *bound* configurations, with the exception of *sim_SS*, achieve essentially the same throughput, because they are executing non-interrupt code on a full-speed complex core.

For this benchmark, core switching imposes a fairly significant cost depending on the configuration of the OS core. The results show that the *V2–V5* kernels generally outperform the *V1* kernel, with the exception of the V2 kernel for architectures *sim_CC* at 1 Gb/s and *sim_Cc* at 10 Gb/s. A trace of the network traffic indicates that both of these experiments retransmit a few packets, which may be the cause. Further, it is not clear for the other experiments whether the differences between the *V2–V5* kernels themselves are consistent.

We also ran this benchmark on the dual-core Xeon hardware, using trials of 15 seconds. Again, the throughput is identical, saturating the 1 Gb/s NIC, independent of kernel configuration. Table 3.9 shows that several different system calls accounted for

Table 3.9. Core-switch counts for 1 Web trial, dual-core X86

Benchmark	Core-switches by system call					
	read	**write**	**open**	**fsync**	**poll**	**sendfile**
Web	2131	7	1829	6	3649	3640

Table 3.10. Simulated Web results on quad-core CPUs. Values are KB transferred during 133 ms

Hardware config.	NIC speed = 1 GBit/sec Kernel configuration						
	Unmod.	**Bound**	**V1**	**V2**	**V3**	**V4**	**V5**
sim_CCSS	2897	2739	685	1187	1222	1239	1227
sim_CCCC	3425	3431	1318	1278	1301	1322	(M5 bug)
sim_CCss	2175	1704	348	1100	1133	1142	1140

Hardware config.	NIC speed = 10 GBit/sec Kernel configuration						
	Unmod.	**Bound**	**V1**	**V2**	**V3**	**V4**	**V5**
sim_CCSS	2937	2795	683	1192	1227	1238	1233
sim_CCCC	3335	3390	1326	(crashed)	1318	1330	1293
sim_CCss	2139	1758	350	1104	1136	1149	1134

the majority of the core switches during this benchmark. *Poll* is normally used to wait for available input; *sendfile* is used to transmit data directly from a file to the network, without copying it into and out of user space. In profiles made on the simulated system, only the *writev* system call consumes significant core time, possibly implying that Apache on the simulated system is not using *sendfile*.

We also simulate quad-core configurations using the notation from Section 3.3, *sim_CCSS*, *sim_CCCC*, and *sim_CCss*, where Apache is bound to the first two complex cores, and system calls and interrupts are bound to the two other cores. The results in Table 3.10 show that in these tests, the fastest core-switching kernel is *V4*, but there is little difference between the *V2*–*V5* kernels.[9] We do not yet solve the scheduling problem for core switching with multiple choices of targets, but profiles show a rough balance of effort between the two OS cores.

[9] One trial failed due to an M5 bug, and another died with a kernel crash, apparently because of a synchronization bug in the core-switching code which allowed a timer interrupt during a critical section.

Table 3.11. Simulated throughput for *ex_tpcb*. Values are transactions/sec. rates (for 100 transactions). This trial used 16 KB L1 caches.

HW	Kernel configuration						
config.	Unmod	Bound	V1	V2	V3	V4	V5
sim_C	11411	NA	NA	NA	NA	NA	NA
sim_CC	10467	12006	7251	3291	3002	11416	11631
sim_SS	4562	4789	4307	4355	4402	4413	4419
sim_Cc	3819	10541	7276	7419	7615	7627	7746
sim_CS	4563	10320	8556	8680	8872	8857	8952
*sim_CS**	5510	11644	9092	9233	9422	9439	9619
sim_Cs	2336	9007	5930	5900	6054	6063	6298
*sim_Cs**	2608	9442	6240	6098	6240	6258	6306

3.6.2 Database Benchmark

The data base benchmark uses the *ex_tpcb* example from the Berkeley DB distribution [96]. Normally, *ex_tpcb*'s throughput is dominated by disk I/O, which makes it hard to evaluate the cost of computation. These experiments eliminate most disk I/O delays by using a RAM disk on the real hardware, and by setting the access time to 1 μs in M5's disk simulator.

Table 3.11 shows the results for trials of 100 transactions, on various simulated hardware configurations. This benchmark requires no warmup period, and only core switches on *fdatasync* and not on other system calls. Even though the application is single-threaded, and actually shows a slight slowdown when going from one to two cores under the vanilla kernel, both the *bound* and core-switching configurations see speedup. We are not sure why the V2 and V3 configurations for *sim_CC* perform so poorly.

Table 3.11 also shows that, in general, the unmodified kernel is the worst performer. This is mostly an artifact of the single-threaded application, which on the unmodified kernel tends to run on core 0. In the experimental configurations, the slower core is numbered 0. When the configuration binds user-mode code to the faster core, this leads to an artificial speedup, except for *sim_SS* and *sim_CC*, which have no faster core. In the cases of *sim_SS* and *sim_CC*, pinning the user code to core 1 has the effect

of separating its execution from the interrupts on core 0, and the modest speedup of *bound* over *unmod* may be the result of more parallelism and/or fewer cache conflicts. Somewhat surprisingly, the V5 kernel usually performs better than the V4 kernel. Perhaps the idle-loop polling in V4 causes interference with the non-idle core.

Table 3.12 shows the results for trials of 20,000 transactions, on the dual-core Xeon hardware. There appears to be no meaningful impact of the core-switching code on this benchmark, even though it spends 33% of its time in the operating system. Table 3.13 shows that essentially all of the core switches happen during the *fdatasync* system call, which flushes a file's kernel buffers to the disk and hence makes a transaction durable.

We modify the M5 simulator to yield procedure profiles for this benchmark. For the uniprocessor (*sim_C*) trial, the system spends 56% of the core in user mode, 24% in system calls (including 13% in *fdatasync*), and 8% in interrupt handlers. For the unmodified kernel on the *sim_CC* dual processor, the primary core spends 53% of its time in user mode, 22% in system calls (12% in *fdatasync*), and 7% in interrupt code; the secondary core was mostly idle.

For comparison, consider the *sim_CS* core with 8 KB L1 caches. For the V5 kernel, the application core spends 42% of its time in user mode, 43% idle, 9% in system calls, and negligible time in interrupts or *fdatasync*. The OS core spends 55% of its time idle, 15% in interrupt code, and 24% in system calls – almost entirely in *fdatasync*. Thus, even though the simple OS core spends more core time executing *fdatasync* than a complex core does, there is enough spare OS core time to maintain throughput, while allowing the high-power complex core to be quiesced in low-power mode for almost half of the time.

Table 3.12. Throughput for *ex_tpcb* on dual-core X86

Core-switching mechanism	$N = 100$ **trials** **20,000 transactions/trial** Mean TPS	Std. dev.	Max. TPS
None	21532	71	21692
V1	21462	78	21623
V2	21467	85	21583
V3	21491	62	21619
V3, work-conserving	21446	75	21576

Table 3.13. Core-switch counts for 1 ex_tpcb trial, dual-core X86

Benchmark	Core-switches by system call: read	write	open	fdatasync
ex_tpcb	81	23	196	200029

Table 3.14. Simulated Netperf results for TCPstream. Values are KB transferred during 167 ms.

Hardware config.	NIC speed = 1 GBit/sec Kernel configuration: Unmod.	Bound	V1	V2	V3	V4	V5
sim_C	21185	NA	NA	NA	NA	NA	NA
sim_CC	21185	21185	21185	21185	21185	21185	21185
sim_SS	13920	19196	14354	14377	13965	14003	13954
sim_Cc	21185	21185	13546	13683	13698	13769	13734
sim_CS	21185	21185	14413	14471	13952	13948	14037
sim_Cs	21185	20939	7139	7128	6990	6973	6990
sim_CS (16 KB L1)	21185	21185	17287	17734	17871	17828	18101
sim_Cs (16 KB L1)	21185	21185	7845	7970	7965	8030	8036

Hardware config.	NIC speed = 10 GBit/sec Kernel configuration: Unmod.	Bound	V1	V2	V3	V4	V5
sim_C	47639	NA	NA	NA	NA	NA	NA
sim_CC	46623	65049	47158	47823	48378	48536	48031
sim_SS	13861	19162	14334	14404	13909	13920	13958
sim_Cc	46180	41563	13468	13680	13728	13778	13746
sim_CS	46185	60718	14421	14484	13951	14002	14014
sim_Cs	46120	23103	7105	7139	6987	6987	7021
sim_CS (16 KB L1)	46172	62680	17357	17700	17766	17827	18046
sim_Cs (16 KB L1)	46146	26547	7836	7953	7965	8002	8039

Table 3.15. Simulated Netperf results: TCPmaerts. Values are KB transferred during 167 ms.

Hardware config.	NIC speed = 1 GBit/sec Kernel configuration						
	Unmod.	Bound	V1	V2	V3	V4	V5
sim_C	17793	NA	NA	NA	NA	NA	NA
sim_CC	17668	16549	16302	16424	16425	16425	16425
sim_SS	17268	16545	16018	16144	16144	16148	16147
sim_CS	17668	16554	16020	16140	16145	16144	16166
sim_Cs	21185	13546	11382	11458	11494	11499	11494

Hardware config.	NIC speed = 10 GBit/sec Kernel configuration						
	Unmod.	Bound	V1	V2	V3	V4	V5
sim_C	20417	NA	NA	NA	NA	NA	NA
sim_CC	20266	18966	18688	18826	18827	18827	18827
sim_SS	18655	19018	17164	17840	17852	17913	17913
sim_CS	20266	18966	17751	17960	17960	18023	18023
sim_Cs	24155	13847	11631	11829	11902	11902	11902

3.6.3 Network Streaming Benchmarks

This section simulates the *netperf/TCPstream* benchmark, which sends TCP data as fast as possible, and *netperf/TCPmaerts*, which receives data as fast as possible. These run for a simulated time of 167 ms, after a 33 ms warmup period. Tables 3.14 and 3.15, respectively, show the results. Except for the *unmodified*, non-bound trials, the application itself is bound to core 1.

In tests with a 1 Gb/s link, *TCPstream* on the unmodified kernel gets essentially full wire bandwidth, except on the *sim_SS* configuration, which with two simple cores seems underpowered. Core switching clearly causes a slowdown on cores with a slow or simple OS core. There is some variation based on the core-switching version.

There is no clear pattern as to which core-switching kernel performs the best. The benchmarks are configured to send and receive data in buffers of 64 KB per system call, which makes core switching relatively infrequent. Perhaps the variation between trials is due to the effects described in Section 3.3.2. The use of core switching, however, does move significant core time from the fast core to the slow core, which reduces throughput.

Also, since almost no time is spent in user mode, the application core is almost entirely idle, and need not be drawing power.

At 10 Gb/s, the simulated architecture could not saturate the network; the core is the bottleneck. Interestingly, on the *sim_CC* configuration, the *bound* configuration yields 40% more throughput than the unmodified configuration. A profile of the unmodified configuration shows that the scheduler puts both the user code and interrupt handling on the same core, leaving the other core fully idle except for clock ticks. The *bound* configuration spreads the load somewhat more evenly over both cores. Again, core switching does cause a slowdown for cores with slower OS cores, and there is no clear winner among the core-switching kernels. Core switching and interrupt binding forces most of the load onto the OS core, again leaving the application core is mostly idle.

With *TCPmaerts*, the results seem inconsistent, possibly because of the effects we describe in Section 3.3.2 as their network traces sometimes show packet retransmission. Profiles show even the uniprocessor core is mostly idle. Table 3.15 therefore omits some rows to save space.

Table 3.16. Core-switch counts for 1 netperf trial, dual-core X86

	Core-switches by system call				
Benchmark	**read**	**write**	**open**	**socketcall**	**poll**
netperf/tcpstream	273	0	120	431216	10
netperf/tcpmaerts	410	1	460	431240	10

On the real dual-core Xeon system, with a 1 Gb/s NIC, multiple trials of both streaming benchmarks always transfer between 941.2 and 941.45 Mb/s regardless of the software configuration, implying that the system is network-limited. We do measure, during the one 60-second trial of each benchmark, the number of times various system calls perform core switches. Table 3.16 shows that almost all of these core switches were in the *socketcall* system call. Linux on x86 differs from Linux on Alpha in that its C

Table 3.17. Energy efficiency comparison between the bound and V5 kernels (KTrans/s/W or MB/s/W)

bench	**os**	**CC**	**CS**	**SS**
Web	Bound	1.31	1.31	7.98
Web	V5	0.87	1.19	5.59
ex_tpcb	Bound	2.31	2.30	9.61
ex_tpcb	V5	2.19	2.79	8.99
netperf/tcpmaerts	Bound	55.75	55.71	363.29
netperf/tcpmaerts	V5	54.96	338.35	354.19
netperf/tcpstream	Bound	47.92	47.82	231.10
netperf/tcpstream	V5	46.89	218.83	225.35

library funnels the socket API through this one system call.

3.6.4 Energy Efficiency

Very fast core switching can save energy by switching from a complex core to a simpler core during a system call. Table 3.17 shows the energy efficiency of the four macro benchmarks for the architectures CC, CS, and SS for both the *bound* and V5 kernels. We compare only these two kernels to separate the impact of parallelism and core switching for energy efficiency.[10] We generate the power information using the McPAT power, area, and timing modeling framework [78]. For each benchmark, either KTrans/s/W or MB/s/W is reported which is equivalent to KTrans/J and MB/J, respectively. Hence throughput per joule is measured for each application, and higher is better.

The SS processor is always more efficient than CC for both kernels, due to the efficiency of in-order execution. The *bound* kernel shows very little difference in energy efficiency between the CC and CS processor configuration. This is because the application is pinned to the complex core, which means that the application code

[10]We avoid the unmodified, unbound kernel because it is hard to separate the parallelism of *specweb* from core switching. We also avoid the core-switching schedulers that use polling and IPI, as this increases energy consumption.

receives the same amount of computational resources on both processors. Also, there is little benefit from running the interrupts on a dedicated complex core compared to a dedicated simpler core. Interrupts for these applications are well suited for the OS core. By comparison to the CC processor, the V5 kernel improves energy efficiency by up to $6.16\times$ for *TCPmaerts* and $3.37\times$ on average on the CS processor.

3.7 Summary

Core-switching costs will become increasingly important with the use of many-core servers. If asymmetry exists between these cores, core switching can migrate code between different cores to improve server energy efficiency. The shorter the delay to core switch, the more often it may be used without violation of server QoS agreements. This chapter shows a series of changes that reduce core-switching costs by half or more to 933 ns compared to the state-of-the-art Linux scheduler. These changes are evaluated using both microbenchmarks and macrobenchmarks on both real and simulated hardware. Core switching on real hardware has no performance impact on macrobenchmarks, because these applications bottleneck on hardware devices like the disk or network. For a simulated system, core switching to slower OS cores on frequent, expensive system calls sometimes reduces performance and sometimes improves performance, but it also provides opportunities to power-down complex application cores, which results in $3.37\times$ greater energy efficiency on average for OS-intense workloads.

Interestingly, the OS-intense workloads that benefit most from core switching either interact with the disk or network. This chapter assumes that the network can always provide the processor with more data to process, which keeps simulation duration tenable and allows close study of the core-switching technique. The next chapter removes this assumption, and considers the network more closely.

Acknowledgments

We could not have done this work without significant help from Rakesh Kumar, Partha Ranganathan, Vanish Talwar, and the members of the M5 community. We are grateful for the helpful suggestions we received from Alexandra Fedorova, David Nellans, and various anonymous reviewers. The research contained in Chapter 3 was supported by NSF grant CCF-0702349.

Chapter 3 contains material from “Fast Switching of Threads Between Cores”, by Richard Strong, Jayaram Mudigonda, Jeffrey C. Mogul, Nathan Binkert, and Dean Tullsen, which appears in *SIGOPS Operating Systems Review*, Volume 43, Issue 2 on April 2009. The dissertation author was the primary investigator and author of this paper. This material is copyright ©2009 by the Association for Computing Machinery, Inc. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Publications Dept., ACM, Inc., fax +1 (212) 869-0481, or permissions@acm.org.

Chapter 4

Integrating Microsecond Circuit Switching into the Data Center

Both power gating and fast core switching from Chapters 2 and 3 can offer improvements in server energy efficiency through hardware-only and software/hardware co-design. However, they both operate under the assumption that servers have access to the data they need to process locally, or that their I/O devices have sufficient bandwidth to keep the processor busy. In contrast to this assumption, sometimes the server spends its time waiting for I/O, especially from the network due to oversubscription. Data centers often use network oversubscription to reduce cost and complexity [1], but pressure to aggregate more services per server [16] in such an environment can lead to poor network performance and increased energy consumption. Oversubscription ratios as high as 240:1 [49] exist in data centers today. They can nearly double the server energy consumption to complete the same amount of work [121] due to increased delay.

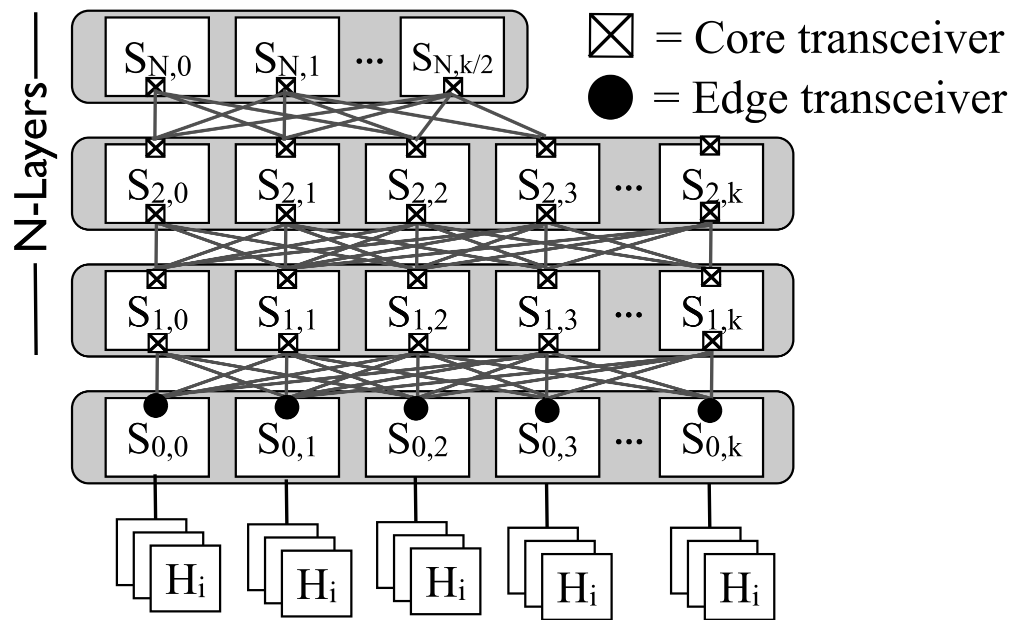
Further, network demand is likely to grow in near future. Servers with 10 Gb/s link rates are common today, and 40 Gb/s NICs are already commercially available. In addition, some big data applications use server memory as a distributed parallel cache to save recently generated data for map reduce like jobs that require several iterations over that data, yielding order of magnitude improvements in performance over

a disk only solution [119].[1] RAMCloud uses DRAM as a replacement for disk to offer up to three orders of magnitude improvement in storage performance [97]. Even the underlying storage for the data center is poised to greatly increase in speed. Today, data centers usually employ the spinning disk technologies that have been around for several decades, and operate near 7,200 RPM. Flash-based SSDs are commodity, scale linearly in performance in RAID arrays, and can offer sequential bandwidths of up to 569 MB/s. Phase change memory could potentially offer an order of magnitude improvement in sequential throughput at 2.932 GB/s [26]. Such an increase in raw storage bandwidth for a distributed file system may put significant pressure on the network.

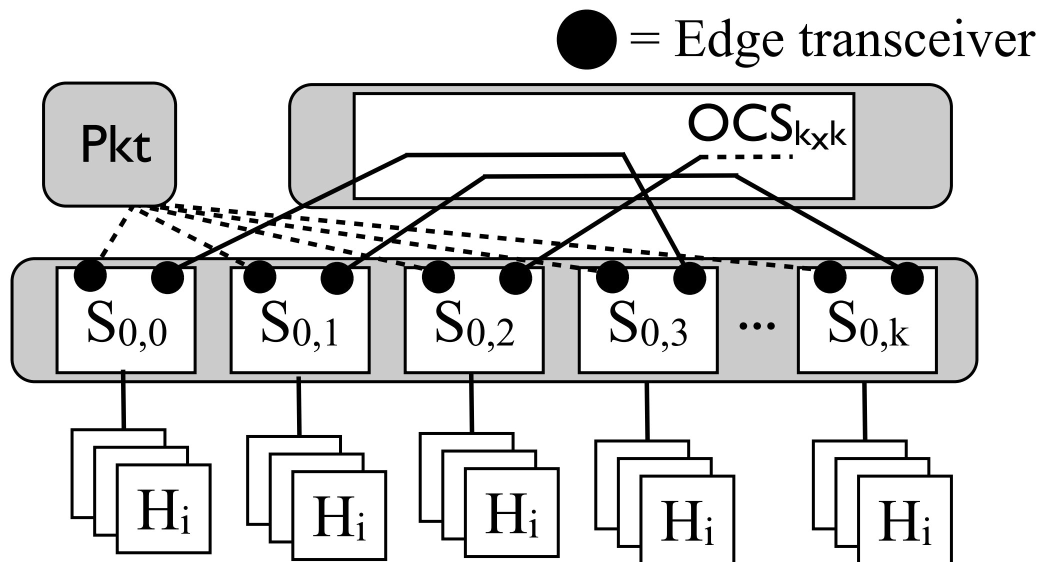
As the size, complexity, and bandwidth demands of data center deployments grow, meeting their requisite bisection bandwidth needs is a challenge. At large scale, this translates into significant bisection bandwidth requirements. To optimize server efficiency for a large data center with numerous, rapidly changing applications, supporting as close to full bisection bandwidth as practical is important. The result is that network complexity and expense are increasing.

To meet the required bandwidth demands, data center operators have adopted multi-layer network topologies [55] (e.g., folded Clos, or "FatTree" [12, 93]), shown in Figure 4.1a. While these topologies scale to very high port counts, they are a significant source of cost and energy, due in part to the large number of switches, optical transceivers, fibers, and power that each of the layers requires. If servers became energy proportional and were only 15% utilized on average, then a full bisection bandwidth network with a peak individual server demand of 1 Gb/s, would consume as much power as the servers [10].

[1] In a personal communication [118], we discover that Spark is often scheduled within a single POD to ensure full bisection bandwidth. However, this technique can have potentially negative consequences on fault tolerance for large distributed jobs, if failure is correlated within the POD. Full bisection bandwidth for the data center alleviates this issue.



(a) A FatTree network topology



(b) A Hybrid network topology

Figure 4.1. A comparison of a scale-out, multi-layered FatTree network and a Hybrid electrical/optical network design. In the FatTree topology (a) each layer of switching incurs additional cost in electronics, core transceivers, fiber cabling, and power. In contrast, the Hybrid topology (b) requires only a single "layer" assuming that the OCS reconfiguration speed is sufficiently fast.

Recent efforts have proposed [29, 43, 115] using optical circuit switches (OCS) to deliver reconfigurable bandwidth throughout the network, reducing some of the expense of multi-layer scale-out networks, shown in Figure 4.1b. A key challenge to adopting these proposals has been their slow reconfiguration time, driven largely by existing 3D-MEMS technology limitations. This reconfiguration time is dominated by two components: (1) the hardware switching time of the 3D-MEMS OCS (10–100 ms), and (2) the software/control plane overhead required to measure the communication patterns and calculate a new schedule (100 ms to 1 s). As a result, the resulting control plane is limited to supporting only highly aggregated traffic at the core of the network [43], or constrained applications with high traffic stability [115].

As optical switches become faster, deploying them more widely in data center networks requires a correspondingly faster control plane capable of efficiently utilizing short-lived circuits. To gain experience with fast OCS switching, we start with supporting a TCP/IP stack across a 24-port OCS prototype called Mordia,[2] which has a circuit reconfiguration time of 11.5 μs. Mordia is built entirely with commercially available components, most notably 2D-based MEMS wavelength-selective switches (WSS). We use this prototype as a stand-in for future low-latency OCS devices.

To enable a TCP/IP stack on top of the OCS, we develop a software top-of-the-rack switch (TOR), capable of acting on circuit reconfiguration events within a microsecond. The software TOR relies on a commodity server and OS module that interact below the TCP/IP stack through softirqs. The software TOR can achieve 65% of the bandwidth of an identical link rate electronic packet switch (EPS) with circuits as short as 61 μs duration, and 95% of EPS performance with 300 μs circuits using commodity hardware. The module is able to run unmodified virtualized infrastructre (Xen and KVM), with support to dynamically switch flows between networks via Open

[2] Microsecond Optical Research Data Center Interconnect Architecture

vSwitch [99]. Taken together, this chapter shows that continuing to push down the reconfiguration time of optical switches, allowing flexible schedules, and reducing the software and control overheads holds the potential to radically lower the cost and energy needed to deliver full bisection bandwidth networks in the data center.

4.1 Related Work

Optical switching technologies: Realistic optical switches that can be used in practice require a limited overall insertion loss and crosstalk, and must also be compatible with commercial fiber optic transceivers. Subject to these constraints, the performance of a switch is characterized by the switch speed and port count. Optical switches based on electro-optic modulation or semiconductor amplification can provide nanosecond switching speeds, but intrinsic crosstalk and insertion loss limit their port count. Analog (3D) MEMs beam steering switches can have high port counts (e.g., 1000 [17]), but are limited in switching speed on the order of milliseconds. Digital MEMs tilt mirror devices are a "middle-ground". They have a lower port count than analog MEMs switches, but have a switching speed on the order of a microsecond [47] and a sufficiently low insertion loss to permit constructing larger port-count OCSes by composition.

"Hotspot Schedulers": This chapter considers optical technology integration in the data center and could be considered complementary to work such as Helios [43], c-Through [115], Flyways [50], and OSA [29], which explored the potential of deploying optical circuit switch technology in a data center environment. Such systems to date have all been examples of hotspot schedulers. A hotspot scheduler observes network traffic over time, detects hotspots, and then changes the network topology (e.g., optically [29, 43, 115] or wirelessly [50]) such that more network capacity is allocated to traffic matrix hotspots and overall throughput is maximized.

Optical Burst Switching: Optical Burst Switching [100, 111] is a research area exploring alternate ways of scheduling optical links through the Internet. Previous and current techniques require the optical circuits to be setup manually on human timescales. The result is low link utilization. OBS introduces statistical multiplexing where a queue of packets with the same source and destination are assembled into a burst (a much larger packet) and sent through the network together. Like OBS, the Mordia architecture has a separate control plane and data plane.

TDMA: Time division multiple access is often used in wireless networks to share the channel capacity among multiple senders and receivers. It is also used by a few wired networks such ITU-T G.hn "HomeGrid" LANs and "FlexRay" automotive networks. Its applicability to data center packet-switched Ethernet networks was studied in [112].

Traffic Matrix Scheduling: A key challenge in supporting microsecond-latency OCS switches is effectively making use of short-lived circuits. Previous works [41, 42] describe an approach to circuit scheduling, called Traffic Matrix Scheduling (TMS). The idea is that traffic demand matrix (TDM) is known for the switch through a demand estimation mechanism. The algorithm massages the TDM into a doubly stochastic BAM via a matrix scaling algorithm [108]. The BAM can then be translated into a schedule of circuits by a matrix decomposition algorithm [21, 114].

Traffic Matrix Estimation: Traffic matrix scheduling, just like hotspot scheduling, requires an estimate of the network-wide demand. There are several potential sources of this information. First, packet counters in the TORs can be polled to determine the traffic matrix, and from that the demand matrix can be computed using techniques presented in Hedera [13]. This method would likely introduce significant delays, given the latency of polling and running the demand estimator. A second potential approach,

if the network is centrally controlled, is to rely on OpenFlow [86] network controllers to provide a snapshot of the overall traffic demand. Third, an approach similar to that taken by c-Through [115] may be adopted: A central controller, or even each TOR, can query individual end hosts and retrieve the TCP send buffer sizes of active connections. Asynchronously sending this information to the TORs can further reduce the latency of collecting the measurements. Finally, application controllers, such as the Hadoop JobTracker [4], can provide hints as to future demands. Our prototype implementation does not implement demand estimation.

4.2 Motivation: Reducing Network Cost via Faster Switching

Decreasing the end-to-end reconfiguration time of optical circuit switching can reduce the cost and complexity of data center networks. We next examine some of the sources of these costs, and motivate the need for low-latency circuit switching.

4.2.1 Multi-layer Switching Networks

While multi-layer switching topologies like FatTrees have been shown to support very large bisection bandwidths, they are a significant source of cost. This cost is likely to increase. To stem this cost increase requires pushing optical circuit switching to a lower layer of the topology — which requires a likewise decrease in OCS switching latency.

Multi-layer packet-switched topologies are very flexible — any node can communicate with any other node on demand. However, they must be provisioned for worst-case communication patterns, which can require as many as five to nine layers in the largest networks, with each subsequent layer less utilized than the next in the common case. Each of these layers adds substantial cost in terms of the switch hardware, optical transceivers, fibers, and power.

Consider a scale-out data center network supporting M servers partitioned into racks (e.g., 20 to 40 servers per rack), and assume that the network is a FatTree. In general, an N-level FatTree built from k-radix switches can support $k^N/2^{N-1}$ servers, with each layer of switching requiring $k^{N-1}/2^{N-2}$ switches (though layer N itself requires half this amount). Therefore, the choice of the number of layers in the network is determined by the number of hosts and the radix k of each switch. Given a particular data center, it is straightforward to determine the number of layers needed to interconnect each of the servers.

There are two trends that impact the cost of the network by increasing the number of necessary layers of switching: fault tolerance and high link rates. We consider each in turn.

Fault tolerance: While a FatTree network can survive link failures by relying on its multi-path topology, doing so incurs a network-wide reconvergence. This can be highly disruptive at large scale, and so redundant links are often used to survive such failures. Dual link redundancy, for instance, effectively cuts the radix of the switch in half since each logical link now requires two switch ports.

High link rates: For mature link technologies like 10 Gb/s Ethernet, high-radix switches are widely available commercially: 10 Gb/s switches with 64 or even 96 ports are becoming commodity. In contrast, newer generations of switches based on 40 Gb/s have much lower radices, for example 16 to 24 ports per switch. Hence, as data center operators build out new networks based on increasingly faster link rates, it will not always be possible to use high radix switches as the fundamental building block. This constraint will necessitate additional switching layers and, thus, additional cost and complexity.

In summary, at large scale several switching layers are likely necessary to deliver scalable bandwidth to a large number of servers. Each layer of switching in the data center network adds additional cost, wiring, and complexity. This cost is driven primarily from three sources: the switches, optical transceivers, and fiber links. Since each layer in a fully provisioned FatTree network consists of $k^{N-1}/2^{N-2}$ switches, these switches constitute a considerable source of cost and motivate increased adoption of OCS switching.

4.2.2 OCS Power Advantages

Optical technologies offer several power benefits in network communications. First, optical technologies require less power than copper to transmit data at a given data rate across the same distance. As the data rate of a copper line increases, so does the loss per meter and required input power to transmit across the same distance of wire. If we consider the Cat6a Ethernet cables suggested to support the 802.3an-2006 10GBASE-T specification, then the loss per meter for a frequency f is approximately $2 * 10^{-5} * f^2$ [87]. At 10 Gb/s, it is possible to transmit 100 m across Cat6a cables, but at 100 Gb/s, the loss and maximum supported Ethernet cable input power limits transmission distance to 7 m. Hence electrical networks in the data center may likely stop scaling past 100 Gb/s. In comparison, an optical fiber suffers loss of only $2 * 10^{-4}$ dB/m [87], separating the concern of transmission rate from distance for today's data center sizes. Hence much less power is needed to transmit optical bits across the data center for current and future data center network speeds.

In addition, the power per port of an OCS network is two and three orders of magnitude lower than an EPS at 10 Gb/s and 100 Gb/s, respectively. For example, the Huawei CloudEngine 12800 Series High-Performance Core Switches can offer 1152 10 Gb/s ports at 14.1 W/port or it can offer 96 100 Gb/s ports at 168.9 W/port [2]. By comparison, a 3D-MEMS Glimmerglass OCS scales at approximately 240 mW/port [43].

Table 4.1. Power consumption of data center networking components. Electrical networking component power consumptions are based on [2]. Optical networking component power consumptions are from [43].

Device	Power
10G-1152 EPS Port	14.1 W
40G-288 EPS port	56.4 W
100G-96 EPS Port	168.9 W
OCS Port	240 mW
Optical Transceiver (λ=8)	1 W
Optical Transceiver (λ=16)	1 W
Optical Transceiver (λ=32)	3.5 W
Fiber	0 W

For core switch components in data centers, it it likely that tens of meters of cabling will be necessary to connect switches from each pod. An EPS based network would need to use fiber with a electrical-to-optical transceiver at both ends of the fiber. An OCS would only need transceivers for EPS ports that uplink to the OCS.

To give a concrete example, we consider a 64 pod, 1024 host per pod, 65,536 server data center similar to [43], where per server peak network bandwidth demand increases from 10 to 100 Gb/s. We assume that the data center network provides full bisection bandwidth in a 2-level multi-rooted tree that connects the uplink ports of each pod to the core switch. The power per port scales according to Table 4.1, and the core switches may be all electrical or optical. We assume that the servers idle at 100 W and reach 250 W at 100% utilization based on in house measurements of server power and the reports of non-proportional server energy in production data centers [16]. Figure 4.2 shows the savings in data center power usage from replacing the EPS core switch layer with an OCS as a function of server utilization. First, we note that as server peak network demand increases from 10 to 100 Gb/s, the average power reduction from an optical switch increases from 7.27% to 23.68%. The major reason for this trend is that the power to support large port counts in an EPS core switch becomes a larger portion of data center

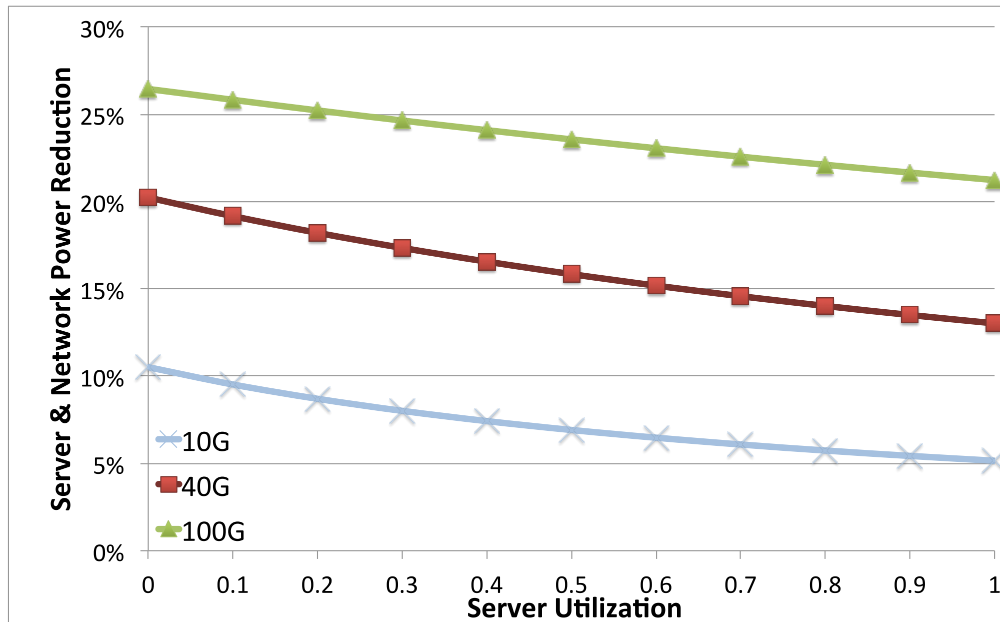


Figure 4.2. Data center server and network power reduction from replacing the core-layer, full bisection bandwidth EPS with an OCS as average server utilization increases. The benefits from an OCS grow for increasing server network demand and lower server utilization.

power at higher data rates. A secondary trend indicates that as servers become more fully utilized, the network's percentage contribution to data center power decreases and dampens the impact of an OCS core switch layer. However, previous studies [16] of data centers indicate that servers are on average 30% utilized, which indicates that an OCS would yield an average power savings between 8.01% and 24.65% for server peak network demands of 10 Gb/s and 100 Gb/s, respectively.

4.2.3 OCS Model

We now describe a simple model of an OCS suitable for supporting a greater share of overall network traffic than previous proposals. This model is similar to that assumed by previous hybrid network designs [29, 43, 115], with a key difference: orders of magnitude faster switching speed.

We consider a model consisting of an N-port optical circuit switch, with a recon-

figuration latency of $O(10)$ μs. Each input port can be mapped to any output port, and these mappings can be changed arbitrarily (with the constraint that only one input port can map to any given output port). The OCS does not buffer packets, and indeed does not interpret the bits in packets either — the mapping of input ports to output ports is entirely controlled by an external scheduler. This scheduler is responsible for determining the time-varying mapping of input ports to output ports and programming the switch accordingly.

For this reason, we assume that TORs attached to the OCS support per-destination flow control, meaning that packets for destination D are only admitted to a switch input port when that input port connects to D. Packets to destinations other than D are queued in the edge TOR during this time. Furthermore, during the OCS reconfiguration period, all packets are queued in the TOR. Since the OCS cannot buffer packets, the TOR must be synchronized to only transmit packets at the appropriate times. This queueing can lead to significant delay, especially for small flows that are particularly sensitive to the observed round-trip time. In these cases, packets can be sent to a packet switch in the spirit of other hybrid network proposals. In this chapter, we focus on the OCS and its control plane in isolation, focusing particularly on reducing the end-to-end reconfiguration latency. In this way, our work is complementary to other work in designing hybrid networks.

4.3 OCS Throughput and Latency

This section compares the bandwidth and latency tradeoffs between an OCS and EPS. An OCS network can leverage greater aggregate bandwidth and lower operational power to offer more efficient full-bisection bandwidth networks. However, because the OCS reconfigures circuits with measurable delay, there is a duty cycle on the bandwidth and an increase in packet latency for those circuits that share a port. The three subsections that follow analyze these tradeoffs in more detail.

4.3.1 Throughput

Compared to an EPS, an OCS can support greater aggregate bandwidth in the network. The reason comes from the difficulty in designing copper cables that support higher data rates, reasonable lengths, and do not exceed the maximal input power that a cable can support as we discuss in Section 4.2.2. By comparison, each fiber of an optical network can use wave division multiplexing to transmit between 44 - 176 channels (depending on channel spacing) at 20 Gb/s [71] per channel yielding between 880 - 3520 Gb/s. Consider the equation for the increase in aggregate bandwidth of on an EPS network as a function of port transmission rate R_{EPS} and number of ports N:

$$B_{EPS} = R_{EPS} \cdot N \tag{4.1}$$

An OCS network scales similarly, except that a duty cycle D is imposed by the duration of each circuit and the reconfiguration time between circuits. Thus, if R_{OCS} is the bandwidth of an an optical port, then the aggregate bandwidth of an OCS switch is given by:

$$B_{OCS} = D \cdot R_{OCS} \cdot N \tag{4.2}$$

The immediate concern is whether the duty cycle negatively impacts OCS throughput. An OCS need only increase its transmission rate or number of supported channels to compensate for a lower duty cycle. Further, it could do these two things without modifying much of the underlaying network with the exception of the transmitters and receivers.

4.3.2 Latency

The delay from sharing a port between two or more circuits could prohibit the introduction of a Mordia-like OCS in the network. To formalize the issue, let us consider the RTT of an OCS:

$$RTT_{OCS} = 4 \cdot \frac{M}{R_{OCS}} + 2 \cdot P_{OCS} + 2W \cdot \frac{C}{1-D} \tag{4.3}$$

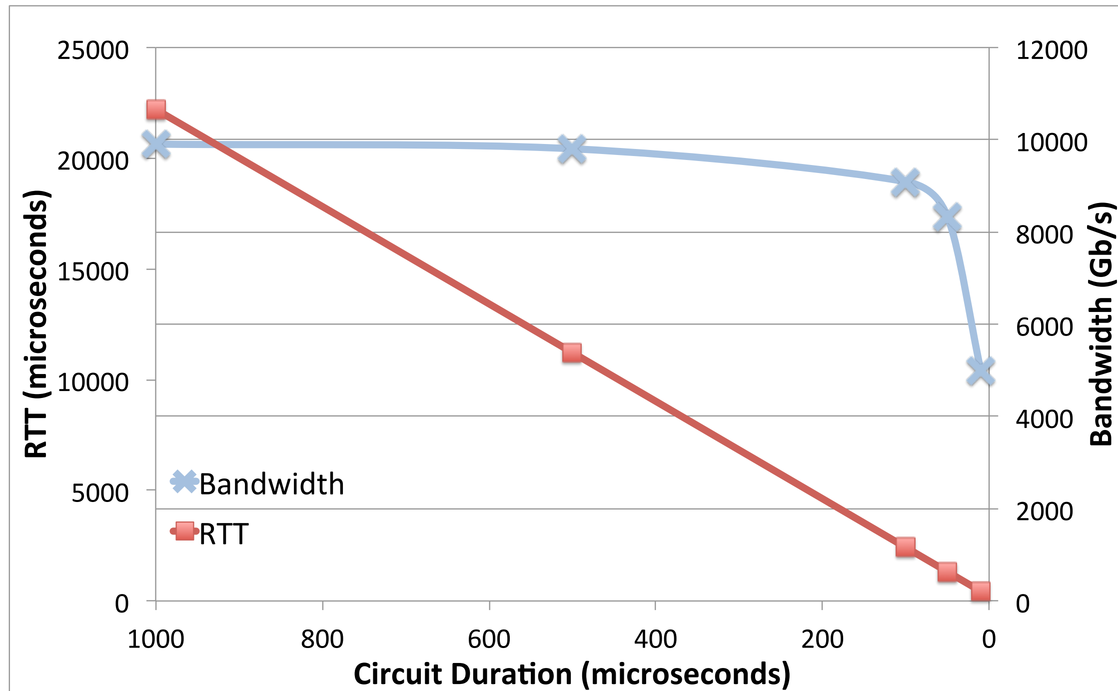


Figure 4.3. The effect of changing duty cycle on OCS bandwidth and RTT

where M is the size of the message, P_{OCS} is propagation delay along fiber, W is half of the average number of circuits that transmit across a port shared with a circuit of interest, and C is the configuration time of the circuit[3]. The average RTT across the OCS is linearly proportional to the duration of the circuit and configuration time ($\frac{C}{1-D}$). However, the greater availability of OCS bandwidth allows the same amount of information to be transmitted in less time. The result is that we can reduce the duration of a circuit (i.e. reduce the duty cycle of a circuit), which more quickly reduces RTT ($2W \cdot \frac{C}{1-D}$) than bandwidth (D) for a significant portion of duty cycle.

To give a concrete example, Figure 4.3 considers the aggregate bandwidth and RTT between two ports of a Mordia-like system with 100 ports, an average of 22 circuits sharing the port interest, 10 μs configuration delay, and 100 Gb/s links as the duration of a circuit ranges from 1,000 μs- 10 μs. At a circuit duration of 1,000 μs, bandwidth is maximized (for this example) to 9,901 Gb/s at the cost of a high RTT time equal to

[3]Equation 4.3 ignores packet processing time

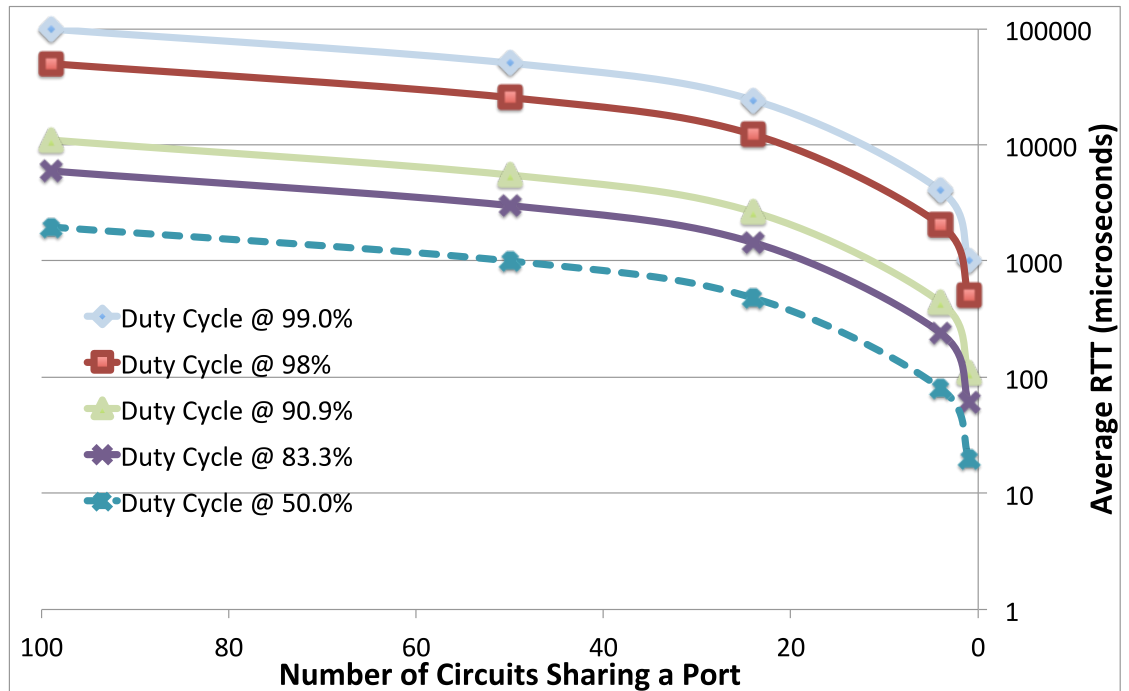


Figure 4.4. The effect of changing duty cycle and number of port sharers on OCS RTT

22,200 μs. However, as the duration of the circuit goes down, the RTT reduces more quickly than the bandwidth (before the knee). At a duty cycle of 50% (i.e. circuit duration is 10us), the OCS can offer 5,000 Gb/s of throughput (5$\times$ the throughput of a 100-port 10G EPS switch) with an average RTT of 440 μs. Thus, a key means of reducing RTT in an OCS is by considering smaller duty cycles with higher bandwidth links and keeping circuit duration short.

When two or more flows require the use of the same input or output port, the result is that those hosts may wait for their required circuit. When we consider Equation 4.3, we realize that on average a circuit waits behind half of the other circuits that currently share a port and need to send packets. Hence any scheme that reduces the number of circuits sharing a port that must transmit simultaneously, also reduces RTT of an OCS network.

Figure 4.4 shows the average RTT for a circuit on a 100 port, Mordia-like OCS,

with 100 Gb/s links, and a configuration delay of 10 μs as the number of circuits transmitting via a shared port decreases from 99 to 1. If we consider a circuit that lasts for 1,000 μs, then the average RTT decreases from 99,990 μs to 1,010 μs as the number of circuits sharing a port ranges from 99 to 1. Further reduction in RTT is possible if this technique is combined with reduction in circuit duty cycle. Figure 4.4 shows what happens to RTT as both duty cycle and port sharing decreases. For instance, a circuit that shares a port with only one other circuit and has a duty cycle of 50% would have an average RTT of 20 μs (assuming negligible packet processing time). Meanwhile, the availability of abundant optical bandwidth would still see throughput benefits for such an OCS.

4.4 Implementation

To evaluate our design, we chose to implement it in a testbed environment. This implementation effort consists of two primary tasks: (1) selecting an OCS capable of switching at O(10) μs, and (2) modifying TORs to support flow control on a per-destination basis at microsecond timescales. Unfortunately, as we discuss in Section 4.1, the only practical commercially available OCSes that can switch in sub-100 μs timescales have small port counts (e.g., 4 or 8 ports). To evaluate at the scale of a TOR (e.g., 24–48 hosts), a prototype OCS is built supporting 24 ports based on commercial wavelength-selective switches, described in Section 4.4.1. Instead of building our own TORs with our flow control requirements, we instead emulate them using commodity Linux servers, as described in Section 4.4.2.

The challenge of selecting an OCS is the main topic of another work [41], and only a brief overview of the prototype is given here. This chapter focuses on the challenge of emulating TORs in software to enable TCP/IP network traffic across an OCS prototype.

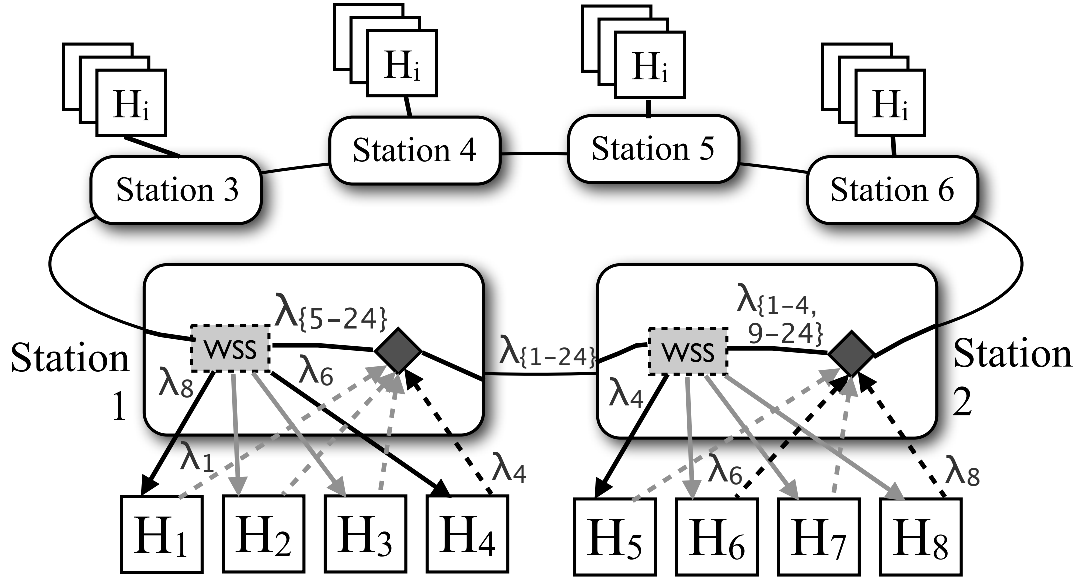


Figure 4.5. The Mordia OCS prototype, which consists of a ring conveying *N* wavelengths through six stations. Each source TOR transmits on its own wavelength, and each station forwards a subset of four wavelengths to the TORs attached to it. This prototype supports an arbitrary reconfigurable mapping of source ports to destination ports with a switch time of 11.5 μs.

4.4.1 Mordia Prototype

The Mordia prototype is a 24-port OCS that supports arbitrary reconfiguration of the input-to-output port mappings. We first describe the underlying technology we leverage in building the OCS, and then describe its design.

Technology

Unlike previous data center OCS designs [43, 115], we chose not to use 3D-MEMS based switches due to their high switch time. The maximum achievable speed of a 3D-MEMS space switch depends on the number of ports. Large port count switches require precise analog control of the 2-axis orientation of relatively large mirrors. Since the mirror response time depends on the size and angular range, there is in general a design tradeoff between the switch port count, insertion loss, and switching speed.

Commercial 3D-MEMS switches support reconfiguration times in the 10s of milliseconds range [3].

Another type of optical circuit switch is a *wavelength-selective switch* (WSS). It takes as input a fiber with N wavelengths in it, and it can be configured to carry any subset of those N wavelengths to M output ports. Typically a WSS switch has an extra "bypass" port that carries the remaining $N-M$ frequencies. We call this type of WSS switch a 1xM switch, and in our prototype, $M=4$. Our switch does not have a bypass port, and so we implement the bypass functionality external to the WSS using additional optical components.

The internal switching elements used in a wavelength-selective switch can be constructed using liquid crystal technology or MEMS [47]. Most MEMS WSSes use analog tilt to address multiple outputs, but at least one commercial WSS has been built using binary MEMS-based switches [110]. Binary MEMS switching technology uses only two positions for each mirror moving between two mechanically stopped angles and also uses much smaller mirrors with respect to a 3D-MEMS space switch. A similar binary MEMS switch is used for commercial projection televisions. The binary switching of small mirror elements results in an achievable switching speed that is several orders of magnitude faster than a commercial 3D-MEMS switch.

In general, there is a tradeoff between 3D-MEMS, which offers high port count at relatively slow reconfiguration time, and 2D-MEMS, which offers microsecond switching time at small port counts (e.g., 1×4 or 1×8). The key idea in the Mordia OCS prototype is to harness six 1×4 switches with bypass ports to build a single $24x24$-port switch. We now briefly summarize the operation of the data path.

Data Plane

The Mordia OCS prototype is physically constructed as a unidirectional ring of $N = 24$ individual wavelengths carried in a single optical fiber. Each wavelength is an individual channel connecting an input port to an output port, and each input port is assigned its own specific wavelength that is not used by any other input port. An output port can tune to receive any of the wavelengths in the ring, and deliver packets from any of the input ports. Consequently, this architecture supports circuit unicast, circuit multicast, circuit broadcast, and also circuit loopback, in which traffic from each port transits the entire ring before returning back to the source. We note that although the data plane is physically a ring, any host can send to any other host, and the input-to-output mapping can be configured arbitrarily (an example of which is shown in Figure 4.5).

Wavelengths are dropped and added from/to the ring at six *stations*. A station is an interconnection point for TORs to receive and transmit packets from/to the Mordia prototype. To receive packets, the input containing all N wavelengths enters the WSS to be wavelength multiplexed. The WSS selects four of these wavelengths, and routes one of each to the four WSS output ports, and onto the four TORs at that station. To transmit packets, each station adds four wavelengths to the ring, identical to the four wavelengths the station initially drops. To enable this scheme, each station has a commercial 1×4-port WSS.

Top-of-the-Rack Switches (TORs)

Each TOR connects to the OCS via one or more optical uplinks. Each TOR internally maintains $N-1$ queues of outgoing packets, one for each of the $N-1$ OCS output ports. The TOR participates in a control plane, which is used to inform each TOR of the short-term schedule of impending circuit configurations. In this way, the TORs know which circuits will be established in the near future, and can use that foreknowledge

to make efficient use of circuits once they are established.

Initially, the TOR does not send any packets into the network, and simply waits to become synchronized with the Mordia OCS. This synchronization is necessary since the OCS cannot buffer any packets, and so the TOR must drain packets from the appropriate queue in sync with the OCS's circuit establishment. Synchronization consists of two steps: (1) receiving a schedule from the scheduler via an out-of-band channel (e.g., an Ethernet-based management port on the TOR), and (2) determining the current state of the OCS. Step 2 can be accomplished by having the TOR monitor the link up and down events and matching their timings with the schedule received in Step 1. Given the duration of circuit reconfiguration is always 11.5 μs, the scheduler can artificially extend one reconfiguration delay periodically to serve as a synchronization point. The delay must exceed the error of its measurement and any variation in reconfiguration times to be detectable (i.e., must be greater than 1 μs in our case). Adding this extra delay incurs negligible overhead since it is done infrequently (e.g., every second).

We use the terminology *day* to refer to a period when a circuit is established and packets can transit a circuit, and we say that *night* is when the switch is being reconfigured, and no light (and hence no packets) are transiting the circuit. The length of a single schedule is called a *week*, and the week lengths can vary from week-to-week. When the OCS is undergoing reconfiguration, each TOR port detects a link down event, and night begins. Once the reconfiguration is complete, the link comes back up and the next day begins.

During normal-time operation, any data received by the TOR from its connected hosts is simply buffered internally into the appropriate queue based on the destination. The mapping of the packet destination and the queue number is topology-specific, and is configured out-of-band via the control plane at initialization time and whenever the topology changes. When the TOR detects that day i has started, it begins draining packets

from queue i into the OCS. When it detects night time (link down), it re-buffers the packet it was transmitting (since that packet likely was 'runted' mid-transmission), and stops sending any packets into the network.

Data plane example

Figure 4.5 shows an overview of the prototype's data path. In this example, there are a three circuits established: one from H_6 to H_4, one from H_8 to H_1, and one from H_4 to H_5. Consider the circuit from H_4 to H_5. H_4 has a transceiver with its own frequency, shown in the Figure as λ_4. This signal is introduced into the ring by an optical mux, shown as a black diamond, and transits to the next station, along with the other $N-1$ frequencies. The WSS switch in Station 2 is configured to forward λ_4 to its first output port, which corresponds to H_5. In this way, the signal from H_4 terminates at H_5. The $N-4$ signals that the WSS is not configured to map to local hosts bypass the WSS, which is shown as $\lambda_{\{1-4,9-24\}}$. These are re-integrated with the signals from hosts H_5 through H_8 originating in Station 2, and sent back into the ring. A lower-bound on the end-to-end reconfiguration time of such a network is gated on the switching speed of the individual WSS switches.

Implementation details

The implementation of the hardware for the Mordia prototype consists of four rack-mounted sliding trays. Three of these trays contain the components for the six stations with each tray housing the components for two stations. The fourth tray contains power supplies and an FPGA control board that implements the scheduler. This board is based on a Xilinx Spartan-6 XC6SLX45 FPGA device. Each tray contains two wavelength-selective switches, which are 1×4 Nistica Full Fledge 100 switches. Although these switches can be programmed arbitrarily, the signaling path to do so has not yet been optimized for low latency. Thus we asked the vendor to modify the WSS

switches to enable low-latency operation by supporting a single signaling pin to step the switch forward through a programmable schedule. As a result, although our prototype only supports weighted round-robin schedules, those schedules can be reprogrammed on a week-to-week basis. This limitation is not fundamental, but rather one of engineering expediency.

4.4.2 Emulating TORs with Commodity Servers

To construct our prototype, we use commodity servers to emulate each of the TORs. Although the Mordia OCS supports 24 ports, our transceiver vendor was not able to meet specifications on one of those transceivers, leaving us with 23 usable ports in total. Each of our 23 servers is an HP DL 380G6 with two Intel E5520 4-core CPUs, 24 GB of memory, and a dual-port Myricom 10G-PCIE-8B 10 Gb/s NIC. One port on each server contains a DWDM 10 Gb/s transceiver, taken from the following ITU-T DWDM laser channels: 15-18, 23-26, 31-34, 39-42, 47-50, and 55-58. Each server runs Linux 2.6.32.

Each of the emulated TORs must transmit packets from the appropriate queue in sync with the OCS at microsecond precision. The source code to our NIC firmware is not publicly available, and so we cannot detect link up and down events in real time and cannot implement the synchronization approach presented in Section 4.4.1. Instead, we modify our prototype to include a separate synchronization channel between the scheduler and the servers that the scheduler uses to notify the servers when the switch is being reconfigured. Ethernet NICs do not typically provide much direct control over the scheduling of packet transmissions. Hence we implement a Linux kernel module to carry out these tasks.

For each emulated TOR, we modify the OS in three key ways to support circuit scheduling while remaining synchronized with the OCS. First, we adapt the Ethernet

NIC driver to listen for synchronization packets from the scheduler so that the host knows the current state of the OCS. Second, we modify the NIC driver to ignore the *link-down* events that occur when the OCS is reconfiguring. Third, we add a custom queuing discipline (Qdisc) that drains packets from queues based on the configuration of the OCS. The subsections that follow start with an overview of the modifications made to the OS, the TCP/IP stack, the Qdisc, and softirqs. We follow this overview with a detailed description of the changes necessary to support a software TOR.

Microsecond TOR Module Overview

Figure 4.6 gives an overview of the emulated microsecond TOR module (MTOR). When a user's application sends data, that data transits the TCP/IP stack (1) and is encapsulated into a sequence of Ethernet frames. The kernel enqueues these frames into our custom Qdisc (2), which then selects (3) one of multiple virtual output queues (VOQs) based on the packet's IP address and the queue-to-destination map (4). The Ethernet frame is enqueued (5) and the qdisc_dequeue function is scheduled (6) using a softirq. The qdisc_dequeue function reads the current communication slot number (7) and checks the queue length (8). If there is no frame to transmit, control is yielded back to the OS (9). If there is a frame to transmit, the frame is DMA copied to the Ethernet NIC (10–12). The total number of packets sent directly corresponds to the number of tokens accumulated in the Ethernet NIC's data structure to control the timing and the rate. The qdisc_dequeue function is then scheduled again (13) until VOQ is empty and control is yielded back to the OS (9). When the next sync frame arrives (14), it is processed, and the scheduling state is updated (15). Then the qdisc_dequeue function is scheduled with a softirq in case there are frames enqueued that can now be transmitted (16). Given that all the packets are only transmitted during the times that the slot is active, the code for receiving packets did not need to be modified. The Mordia Qdisc details are expanded in

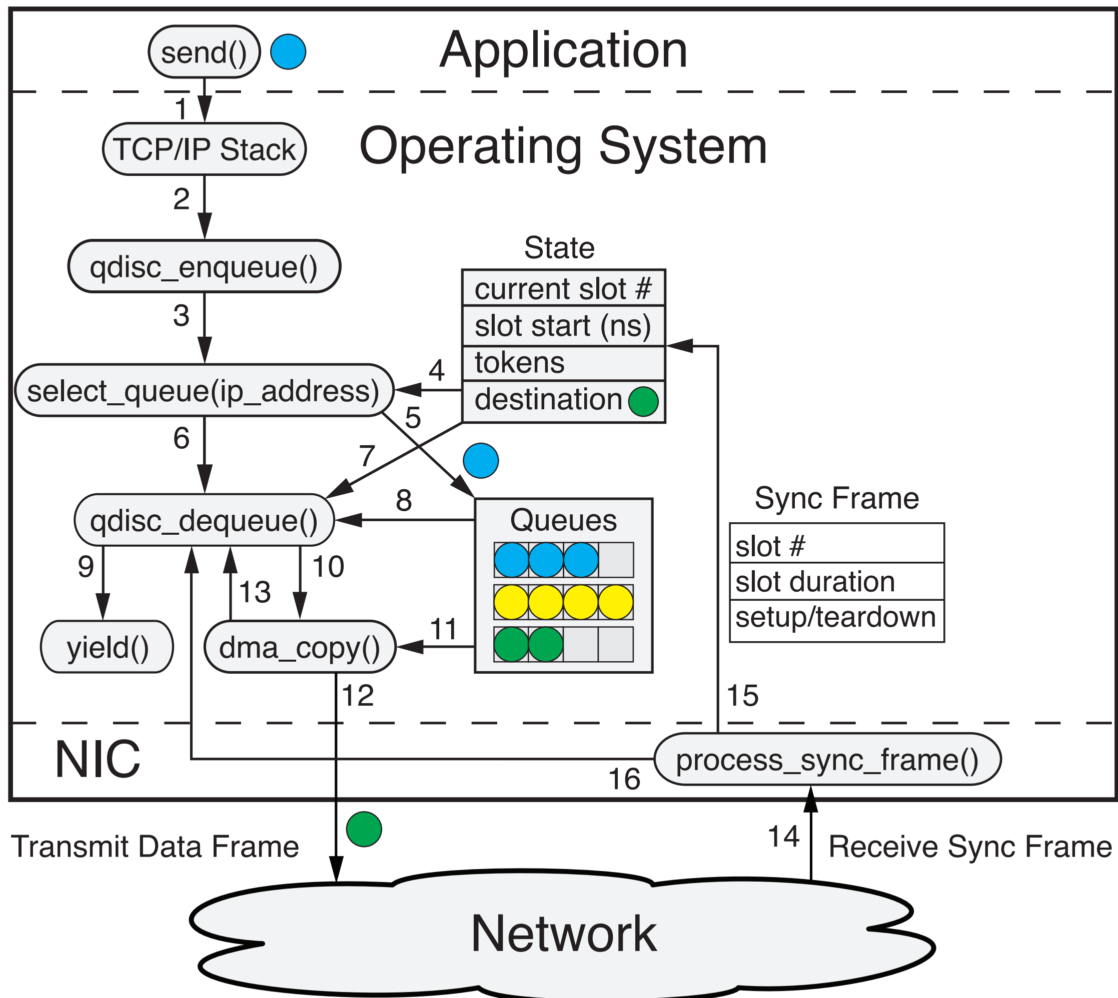


Figure 4.6. A software implementation of multi-queue support in Linux using commodity Ethernet NICs. Sync frames coordinate state between each emulated TOR (server) and the scheduler, so that each Qdisc knows when to transmit Ethernet frames.

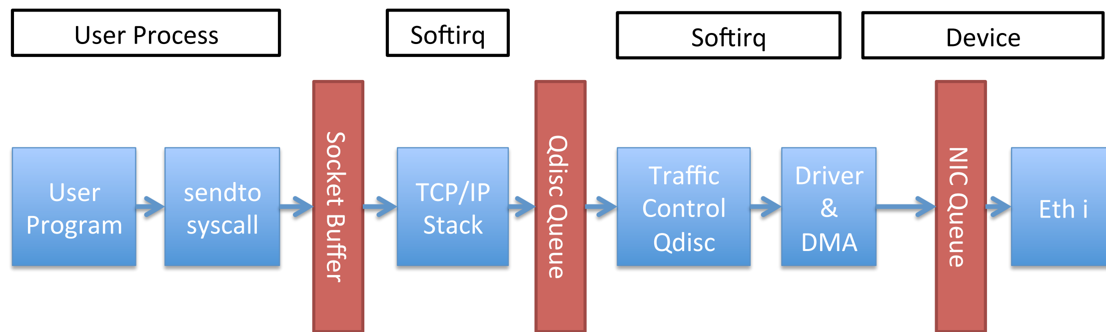


Figure 4.7. Steps for a user program to transmit a packet.

the the later sections.

Link-Down Events

Since the OCS is switching rapidly, the host NIC ports attached to the OCS experience numerous link-up and link-down events. When Linux receives a link-down event, it normally disables the interface, resetting and closing any open sockets and TCP connections. To prevent these resets, we disable the link-down and link-up calls in the NIC driver. Our NIC vendor believes that with access to the firmware source code, synchronization could be implemented entirely in hardware, obviating the need for ignoring link-up and link-down events.

A Packet's Stack Life & Sources of Delay

Figure 4.7 shows the steps necessary for a user-space program to transmit a packet. The user-space program allocates a socket, transmits data with the sendto system call which adds the data to a socket buffer. The TCP/IP stack is then processed and the packet is added to a Qdisc traffic control queue. A softirq then dequeues packets from the Qdisc traffic control layer and delivers the packet to the NIC driver, which DMAs the packet to the NIC for transmission. In order to support low-latency circuits, the behavior of stages past the Qdisc queue are critical. For instance, if the traffic control Qdisc is too slow, the maximum data rate that the Qdisc can deliver to the driver may be significantly

slower than the data rate of the NIC. Further, if the latency is too high between when the driver receives the packet and the packet is transmitted along the link, then high data rates may be supported with the negative consequence that the packets are transmitted along the wrong circuit or while the circuit is currently inactive.

To make matters worse, the stages past the Qdisc queue are subject to variability in processing time. The Qdisc traffic control system is handled by softirqs subject to preemption by interrupts, including the timer interrupt, which may indicate that the softirq has run too long, and that another process should be allowed to run [19]. Further, many devices use softirqs (or their derived tasklet form) to handle the bottom half of the interrupt. Thus, softirqs may need to share processor time with other softirqs. In our experience, this variability can cause unexpected delays in processing a softirq from less than a microsecond[4], to as much as several hundreds of microseconds.

The network card is also subject to variability. In particular, the variability in delay between when a packet is sent to the NIC for transmission and it appears at the neighboring NIC increases as the data rate increases.[5] We found that below 8.2 Gb/s, the Myricom 10GE NIC experiences between 1 - 23.5 μs delay, as a monotonic function of packet size. However, as the data rate at which packets transmit increases up to 10G, the delay increases to as much as 100 μs later at the receiving host. Should we add packets to the NIC faster than 10 Gb/s (i.e. we partially fill the NIC buffer), it may take as long as 400 μs for those packets to appear at the neighboring host. The result is that when the OS delivers packets to the NIC at higher rates, the delay and jitter seen from the NIC increases. Section 4.5.1 discusses experimental results about this delay.

[4] By less than a microsecond, we mean that we could detect no delay from the linux kernel 2.6.32 ktime_get() function.

[5] On first consideration, the increase in delay may seem to be the result of queuing delay. However, there is a considerable 10-300 μs jump in delay as soon as we queue 80 KB to the NIC queue. We suppose that a combination of queuing delay and some additional hardware path are combining to increase packet delay.

Last, the user application experiences a delay from the TCP/IP stack. To test this, we perform two echo experiments. The first echo experiment uses a traditional linux socket, while the second echo experiment uses a kernel bypass API provided by the Myricom sniffer library [7]. The Myricom sniffer library bypasses the kernel's involvement in packet transmission, allowing a user application to deliver a char pointer array directly to the NIC for transmission. Hence this experiment allows one to measure the overhead of the kernel's transmission and reception stack. This measurement found that the average overhead of the kernel's transmission and reception averages 10 μs.

OS Pre-Driver Packet Handling

If a device raises an interrupt, the OS divides processing of that interrupt into two halves [82]. The *top half* saves important device state the OS requires to process the interrupt, and sets a flag indicating that more processing is required. The OS then processes the *bottom half* at a later time when checking the flags set by the top halves. The bottom half is responsible for handling the greater share of device information processing. The reason for this model lies in the state of the system when a *top half* runs. The *top half* runs with interrupts disabled and may not be preempted by another process. Thus, too much processing in the *top half* impairs the OS's ability to respond to further interrupts. In contrast, *bottom half* handlers run without interrupts which allows the OS to preempt them with other interrupts, or time share with other processes.

Both reception and transmission of packets use the two-half protocol and since the 2.4 kernel, softirqs have been the preferred *bottom half* handlers. For packet reception, the *top half* interrupt handler copies the packet into a sk_buff data structure usually via DMA, initializes the sk_buff and device structures, adds the device to a list of devices that currently have packets, and then sets the flag for the NET_RX_SOFTIRQ sotirq. At a later time (i.e. often right after packet reception), the OS handles the NET_RX_SOFTIRQ

by calling the *net_rx_action* function which is responsible for going through the list of devices that currently have packets, processing those packets per device subject to a total packet processing budget, and a time limit of one scheduling interval.

Packet transmission goes through a similar sequence, with the exception that the OS processes the packet with several softirqs before the OS hands it to the driver. Once a packet makes it through the network protocol stack, the timing critical steps occur for transmission. Within a single softirq, the function *dev_queue_xmit* receives a packet, enqueues the packet to the kernel's queuing discipline (Qdisc), then tries to dequeue a packet from the Qdisc. If *dev_queue_xmit* dequeues a non-null packet, it grabs the NIC driver's lock, gives the packet to any registered sniffer (e.g. *tcpdump*), and then delivers the packet to the driver for transmission.

In between the steps when *dev_queue_xmit* enqueues and dequeues a packet lies the netfilter's Qdisc. The Qdisc is a powerful addition to the Linux kernel's traffic control system that gives a user a hierarchal means of enforcing different policies and constraints on network packets [60]. The Qdisc is able to control which packets are enqueued and dequeued during each packet transmission, in addition to passing a subset of the packets to other Qdisc's to enforce additional policies. In its simplest form, a Qdisc may use the pfifo_fast queuing discipline that provides three FIFO queues with monotonic priorities. The OS enqueues and dequeues packets from the context of the softirq which is discussed above. The primary importance of the Qdisc is that it allows the OS to enforce arbitrary policies on network packets below the level of the user application and network stack. Hence the Qdisc is well suited to adapt to low-latency circuits without modification to user applications.

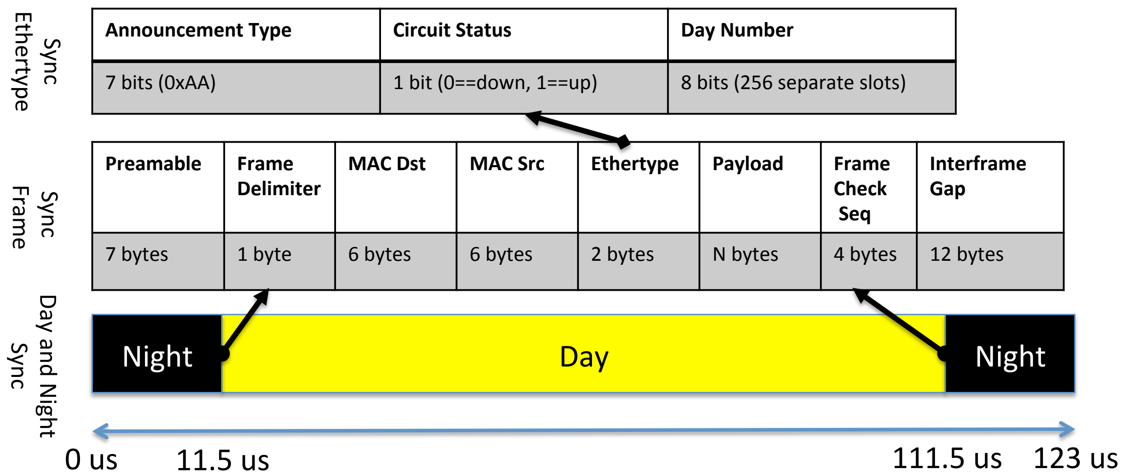


Figure 4.8. Organization of day and night synchronization frames. The synchronization frame holds the day duration information in the payload.

Synchronization Packets

The OCS FPGA controller transmits synchronization packets to a separate 10G packet-switched network which connects to a second Ethernet port on each server. In order to participate with the OCS prototype, each server must listen for Ethernet frames with a protocol type of 0x8015. The frames come in pairs at least 11.5 μs apart. The first frame, called a *night* sync, announces that a new circuit is being setup and that the link is *down* (i.e. no light may transmit through the link). This means that the server should immediately stop transmitting packets meant for the previous circuit or risk losing packets. The second frame, called a *day* sync, is almost identical to the *night* sync except that the circuit state is set as *up*, notifying the servers that packets may successfully transmit along the link. The synchronization frames include the day number, the day's duration, and whether the circuits are being setup or torn down (see Figure 4.8). [6]

[6] We overload the Ethertype field, because our previous experience with overloading the MAC source address causes the EPS to fail after learning too many mappings.

Kernel Modifications for a Microsecond TOR

The Microsecond TOR (MTOR) Qdisc separates the user application's request to send data from how the OS delivers data to the correct circuit. The user application can pass a char buffer of data to the socket indifferent to both the data's destination and circuit availability. The MTOR Qdisc enqueues data into per destination VOQs, until a demanded circuit is active. Once a circuit is active, the MTOR Qdisc dequeues data from the VOQ associated with an active circuit.

We design the MTOR Qdisc to minimize core utilization when there is no data to send, and to minimize latency when packets wait in at least one VOQ. Overall, the design resembles the NAPI interrupts design [19] in its use of event triggered polling. The only two times when the MTOR Qdisc would be scheduled to run is (1) when the receive-side softirq detects a new sync frame announcing a circuit for a VOQ with more than zero packets or (2) when the MTOR Qdisc dequeus a packet for a VOQ which has more than one packet. Beyond these two conditions, the MTOR Qdisc remains unscheduled.[7] We now go over the OS modifications necessary to support a VOQ enqueue, the processing of sync frames, and event based dequeue.

Once a packet transits the TCP/IP stack, it enters the MTOR Qdisc enqueue function (see Figure 4.9). Initially, the MTOR Qdisc assigns each packet to the DROP VOQ where packets are dropped instantly. If the packet is an IP packet, the MTOR Qdisc checks a lookup table for a mapping of the IP address to VOQ number.[8] If no mapping exists, the MTOR Qdisc drops the packet; otherwise, the MTOR Qdisc assigns the packet to a VOQ. Then, the MTOR Qdisc checks the assigned VOQ for space to hold the packet. If the VOQ is full, the MTOR Qdisc drops the packet and notifies the higher levels of the TCP/IP stack to temporarily stop transmitting packets. If the VOQ is not full, the MTOR

[7] The OS still periodically schedules the Qdisc every few seconds, to avoid a blocked packet

[8] Should the circuit connect to a range of hosts, it would be straightforward to queue packets into range queues.

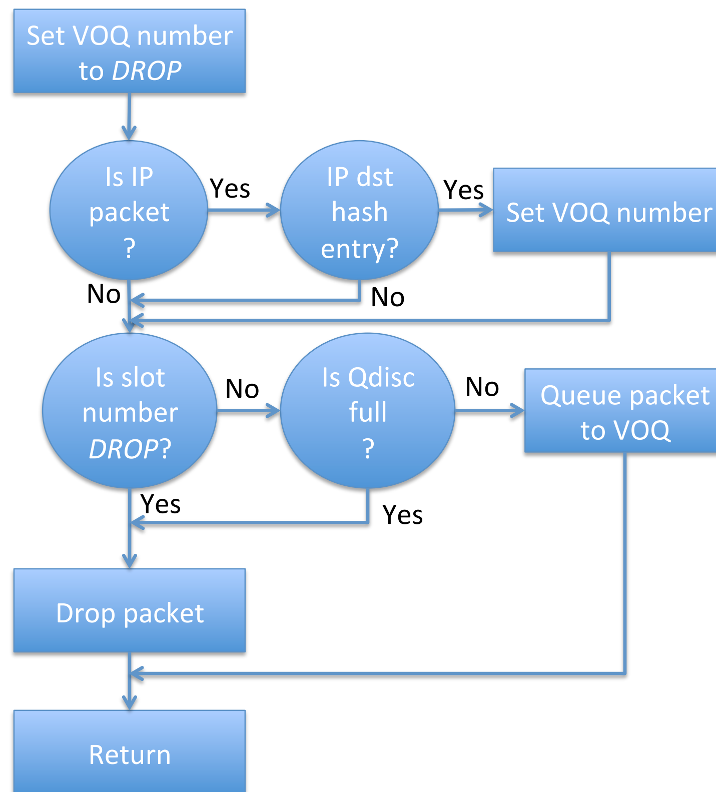


Figure 4.9. A flow diagram depicting the MTOR Qdisc enqueue function

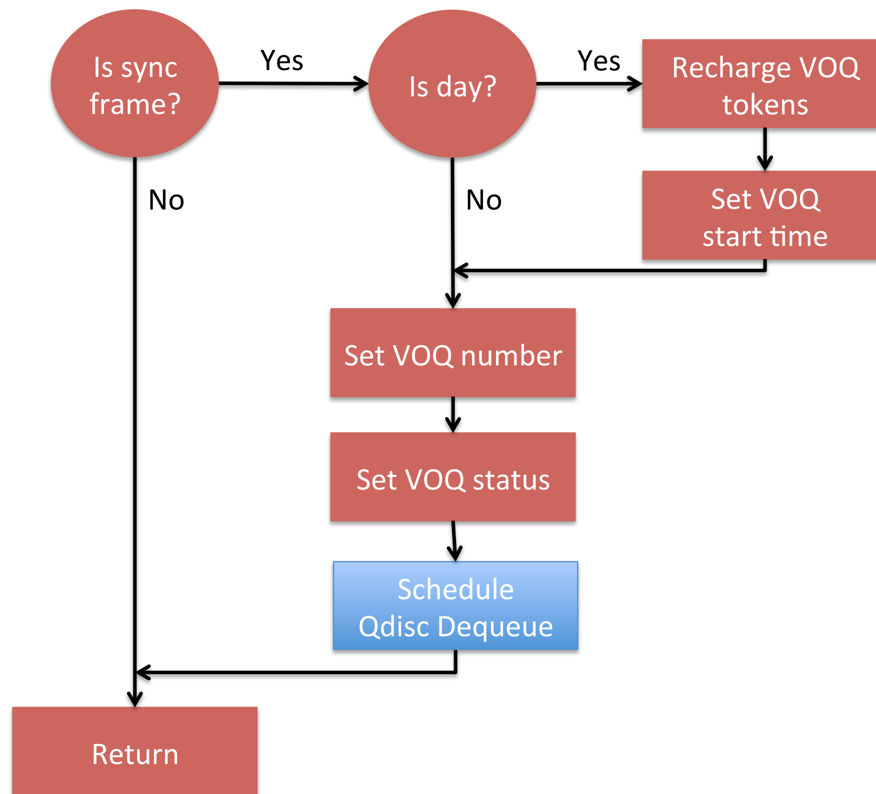


Figure 4.10. Pseudo-code flow diagram of the modifications to the OS NIC driver to support low-latency processing of synchronization frames.

Qdisc enqueues the packet to the VOQ for later transmission.

In order to know when to dequeue packets from a particular VOQ, the OS monitors received packets for sync frames, and Figure 4.10 depicts the pseudo-code flow diagram of the changes made to the OS to support processing synchronization frames. When the OS receives a packet, it processes that packet in an receive-side softirq. We modify this softirq to check the Ethernet frame protocol type for 0x8015. If the OS matches the frame, then it decodes the packet and updates a shared-memory table that maintains the current circuit and the circuit's status (*day* or *night*). If the circuit's status is set to *day*, the OS recharges the tokens for that circuit, and records the time when the circuit becomes active. We use the start time of each circuit to add a safety check to

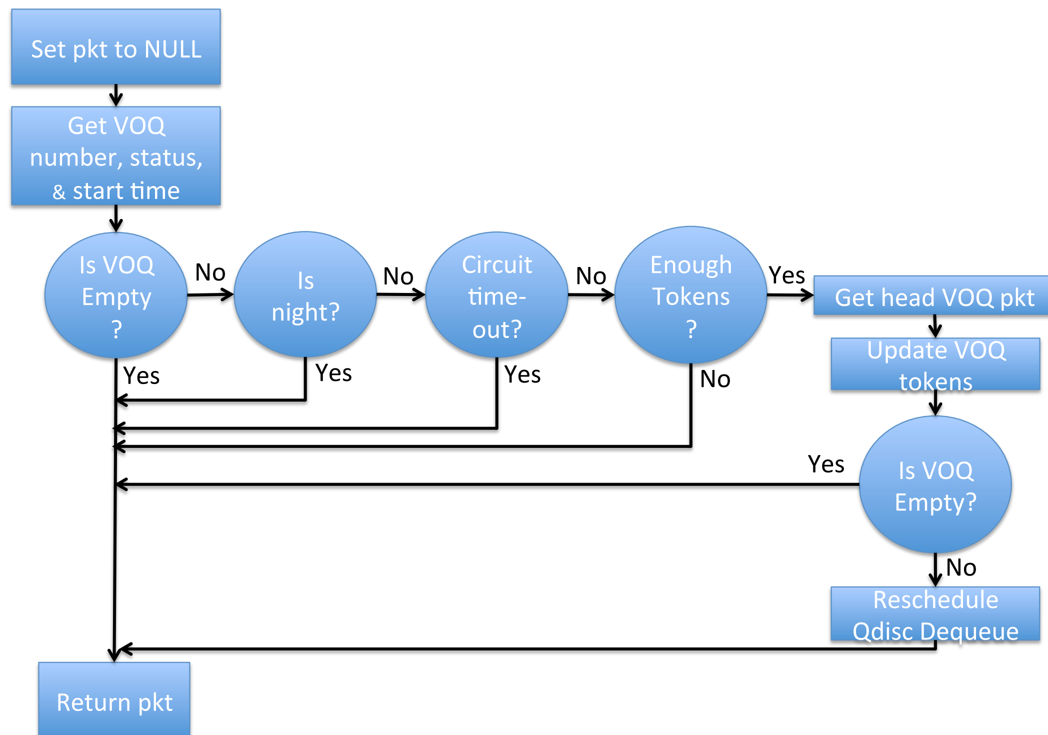


Figure 4.11. A flow diagram depicting the MTOR Qdisc dequeue function

ensure that the OS does not transmit packets at *night*.[9] After processing the sync packet, the OS schedules the MTOR Qdisc to dequeue packets (the code depicted in Figure 4.11). In this way, the MTOR Qdisc only attempts transmit packets when the status of the circuit connections changes.

If the OS detects a sync frame indicating that a VOQ with more than zero packets may start dequeuing packets, it schedules the MTOR Qdisc dequeue function (see Figure 4.11). This function dequeues a packet from the active VOQ as quickly as possible and attempts to maximize the probability of successful reception. First, the function checks the shared-memory table that saves the currently active VOQ, the circuit's status, and the time at which the circuit becomes active. Next, the MTOR Qdisc performs four checks to ensure that a packet should be transmitted. (1) If the MTOR Qdisc detects

[9]Although not shown in the flow diagram, the Qdisc also checks the timestamps for anomalous sync frame timings which may indicate a problem in the system

that the VOQ is empty, then it returns a NULL packet to indicate to the OS that there is no data to send, and the MTOR Qdisc does not run until the OS receives a new sync frame. (2) If the current circuit is in the night state, the MTOR Qdisc returns a NULL packet. (3) The MTOR Qdisc checks the offset within the day of the circuit to ensure that the packet is not transmitted at the expected night interval. (4) The MTOR Qdisc also checks the total remaining tokens to ensure that the packet is allowed to transmit. If the MTOR Qdisc passes all of these checks, then it dequeues the head packet from the VOQ and updates the VOQ's tokens. If the VOQ has more packets to send, the MTOR Qdisc dequeue function reschedules itself. Rescheduling dequeue is critical for performance as without it, the dequeue function would only be called on processing new sync frames and right after enqueue.[10] Last, the MTOR Qdisc returns the packet it dequeues from the head of the active VOQ, and the OS delivers the packet to the NIC driver.

Enhanced Token Control

The MTOR Qdisc dequeue function checks the total number of tokens available to the current VOQ before any packet may be transmitted. A naive token scheme might only allow the NIC to transmit $D*R$ bytes across a circuit, where D is the duration of the day in seconds and R is the link rate of the NIC in bytes per seconds. The naive scheme would assign a number of tokens equal to this number of bytes for each active VOQ.

One problem with this scheme occurs when a large group of packets become ready for transmission at the end of the day, a situation named *evening rush*. *Evening rush* can cause the NIC to queue up more packets than the NIC can transmit with the remainder of the day's duration. This situation happens because the OS can queue up packets to the NIC much faster than the NIC can transmit them. Figure 4.12 considers

[10]The reader may wonder when the MTOR Qdisc enqueue function calls the dequeue function as it is not shown in Figure 4.9. The call to the Qdisc dequeue function occurs right after the OS calls Qdisc enqueue. The code is in the vanilla kernel and hence is not a part of the OS modifications.

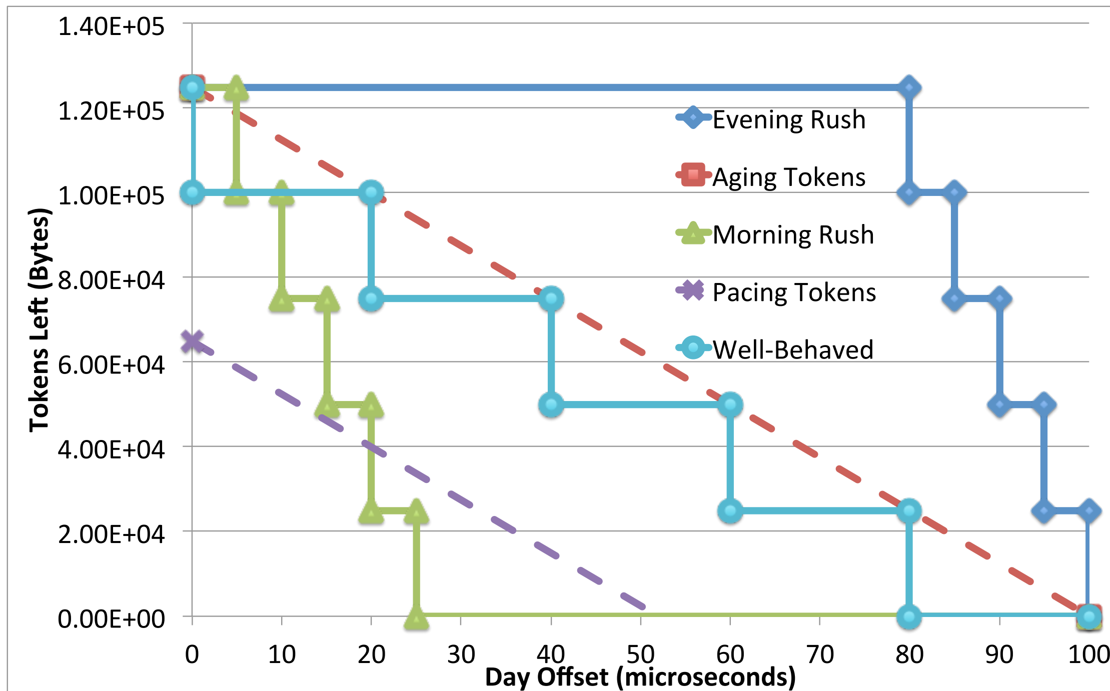


Figure 4.12. The case for enhanced token management

this scenario for a circuit configuration with 100 μs days for a connection that supports a link rate of 10 Gb/s (125,000 bytes may be transmitted in 100 μs). The way to avoid this problem is to cause tokens to *age* throughout the day in such way that for a day with δ seconds remaining and a link rate of R bytes per second, only $\delta * R$ bytes may transmit.

The less intuitive problem occurs when a large group of packets all demand to be transmitted at the beginning of the day, a scenario named *morning rush*. *Morning rush* can cause a large delay in packet transmission of up to 300 μs resulting in average packet loss rates of 2.2%. The issue grows grows worse as days increase from 61 μs to 1 *ms*. Once the MTOR Qdisc queues more than 80 KB of data to the NIC, the NIC enters a longer latency batch mode that is optimized for throughput as opposed to latency. To resolve this issue, we track the difference between the number of age tokens and the total tokens left for the VOQ to transmit pakcets. If the difference grows above 80 KB, the MTOR Qdisc dequeue function reschedules itself and returns zero tokens. We call this

```
int
tokens_left(struct Qdisc *sch, struct mtor_voq *voq, int day_offset)
{
    int age_tokens;
    mtor_sched_data *q = qdisc_priv(sch);

    // compute the number of tokens left from aging
    age_tokens = voq->token_recharge
        - (voq->token_recharge/q->us_per_day)*day_offset;
    voq->tokens = MIN(age_tokens, voq->tokens);

    // compute the number of tokens left from pacing
    if ((age_tokens - voq->tokens) > 80E3){
        // raise softirq to call Qdisc dequeue
        // after other pending softirqs
        __netif_schedule(sch);
        return 0;
    }
    return voq->tokens;
}
```

Figure 4.13. Actual OS code for how the *enough_tokens* function stage in Figure 4.11 determines the number of tokens left

scheme *pacing* tokens. Figure 4.12 summarizes the two problems with the naive scheme and the impact of enhanced token management.

Figure 4.13 shows the OS code that computes the total number of tokens. Using this code, 23 servers, and 506 GB of random data transmitted in an all-to-all pattern via UDP, we characterize the improvement that the different token schemes have on packet loss rate. Figure 4.14 shows the average loss rate per host as a function of day duration ranging from 61 to 300 μs for three token schemes. The naive scheme losses an average of 1.79% packets. The *aging* token scheme fairs somewhat better losing 1.65% packets on average. The token scheme that employs both *aging* and *pacing* losses only 0.44% packets on average, 4$\times$ less than the naive scheme. Interestingly, the loss rate goes down as day length increases from 200 to 250 μs for both the naive and *aging* scheme. This occurs because the duration of the day begins to start matching the increase in delay of

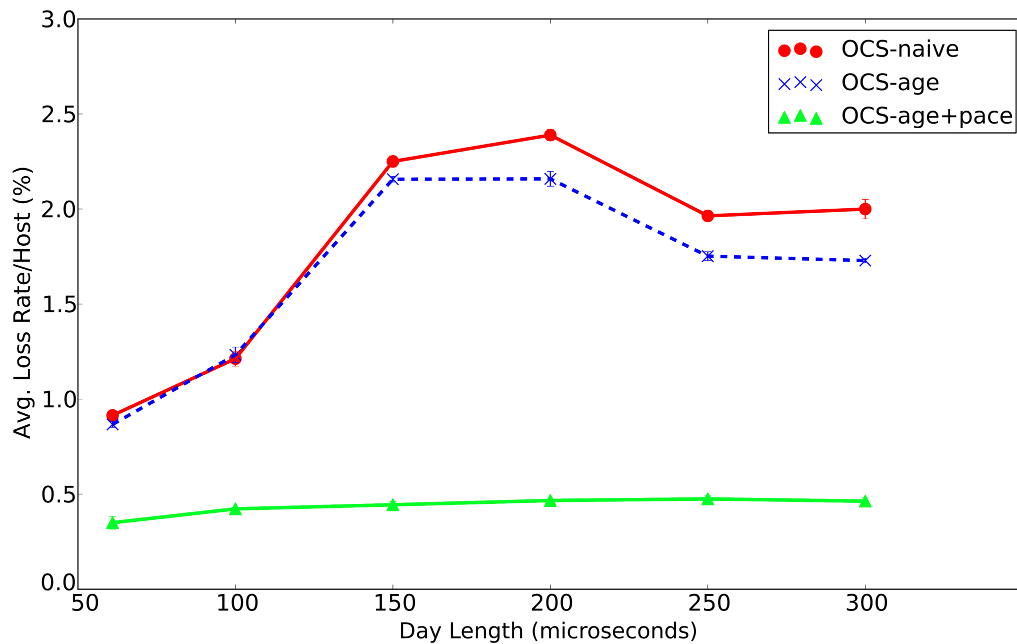


Figure 4.14. The impact of aging and pacing tokens on packet loss rate. Measurement standard deviation is shown as verticle bars, that are sometimes obstructed by the plotted dots when the deviation is small.

the NIC batch mode.

Server Configuration

In order to improve performance and reduce latency of the MTOR, we set several system options. First, given the importance of cooperation of interrupts and softirq's to the MTOR's performance and the high data rates of the NICs (e.g. 10 Gb/s), we gave special care to their settings. The NIC devices use NAPI interrupts [19], which combine the benefits of interrupts and polling within a single scheme. At low loads, the NIC signals new packet arrivals and transmissions to the processor with interrupts. However, if the NIC receives or transmits new packets before the OS processes the previous packet, the NIC driver avoids setting an additional interrupt because the OS polls for additional packets until the queue is empty.

To prevent one core from being a bottleneck for packet transmission and reception,

we configure each NIC interface to use two separate queues in which packet transmission and reception may occur on either queue. The OS assigns a separate interrupt to each queue, and binds the handling of interrupts associated with the separate queues to distinct cores. For our experiments, care is taken to prevent user processes from running on the NIC interrupt cores via the taskset command. This implies that our current configuration sacrifices two cores for efficient use of circuits. However, the fact that our server has 16 cores available and the promise of Moore's Law, minimize concern for core utilization.

The Myri-10G Dual-Protocol NIC offers two features to improve performance in network code that prove to be problematic in our timing sensitive code, which we disable. First, the NIC offers the capability to coalesce interrupts for some period of time in microseconds. During this period, the NIC does not raise an interrupt for incoming packets. At the end of the period, the NIC raises an interrupt if it receives at least one packet at which point the NAPI polling mechanism processes the packet. This coalesce delay adds to the processing latency of sync frames and also leads to data packets being time stamped later than they are actually received.

Second, the NIC offers support to offload TCP segments (TSO). TCP TSO allows the network stack to pass TCP segments much larger than the maximum segment size. The NIC then sends the segments in MTU sized chunks and reassembles the segments at the receiving host [88]. The problem is that TSO can delay the transmission of TCP segments, which can result in TCP traffic blasting data into an inactive circuit connection after the *night* sync. In addition, TSO causes received packets to get timestamps tens of microseconds later than when they are actually received.

Finally, we modify the default configuration of several TCP parameters in the TCP/IP stack. We find that setting the initial congestion window and receive-side buffer of TCP so that at least two days worth of packets can saturate each connection maximizes TCP performance, similar to the Blast protocol of [124]. We also configure the OS's max

packet backlog to support 23 simultaneous 10G connections, as otherwise, the kernel drops many packets.

4.4.3 All-to-All Traffic Generator

In order to fully exercise all-to-all connectivity with N hosts, we design a custom benchmark such that each host blasts traffic to N-1 other hosts as quickly as possible for both the UDP and TCP protocols, named *a2a-syngen*. Each host spawns a client and server process that is responsible for transmitting and receiving traffic. The UDP server allocates a single socket, polls that socket for new data, and sums up the total data in bytes per each IP source. The UDP client pre-allocates N-1 non-blocking sockets pointing to the N-1 hosts, respectively, and a data buffer H bytes smaller than the MTU, where H is the number of header bytes in order to send full Ethernet frames with preset data. When ready to send, the UDP client attempts to transmit a packet to each host in round-robin fashion. If an attempt to send a packet on a socket fails, the socket does not block and retries on the next transmission round. To support the TCP protocol, the TCP server allocates N-1 sockets to each remote host and uses the poll system call to check for data on any socket. Each TCP connection with the N-1 hosts is pre-established before the experiment to avoid measuring TCP setup time. The TCP client operates similarly to the UDP client with the exception that each socket must connect with the server before the experiment may proceed.

One additional server synchronizes the N hosts by requiring each host to go through five phases which include: *AWAKE*, *SETUP-SERVER*, *SETUP-CLIENT*, *RUN-EXPERIMENT*, and *CLEANUP*. The synchronization host, S, uses *parallel-ssh* to wake up clients on all N hosts. The clients respond that they are *AWAKE*. Next, S sends a command which causes all hosts to enter the *SETUP-SERVER* phase during which the hosts load kernel modules, initialize the interface, test for connectivity, start the server

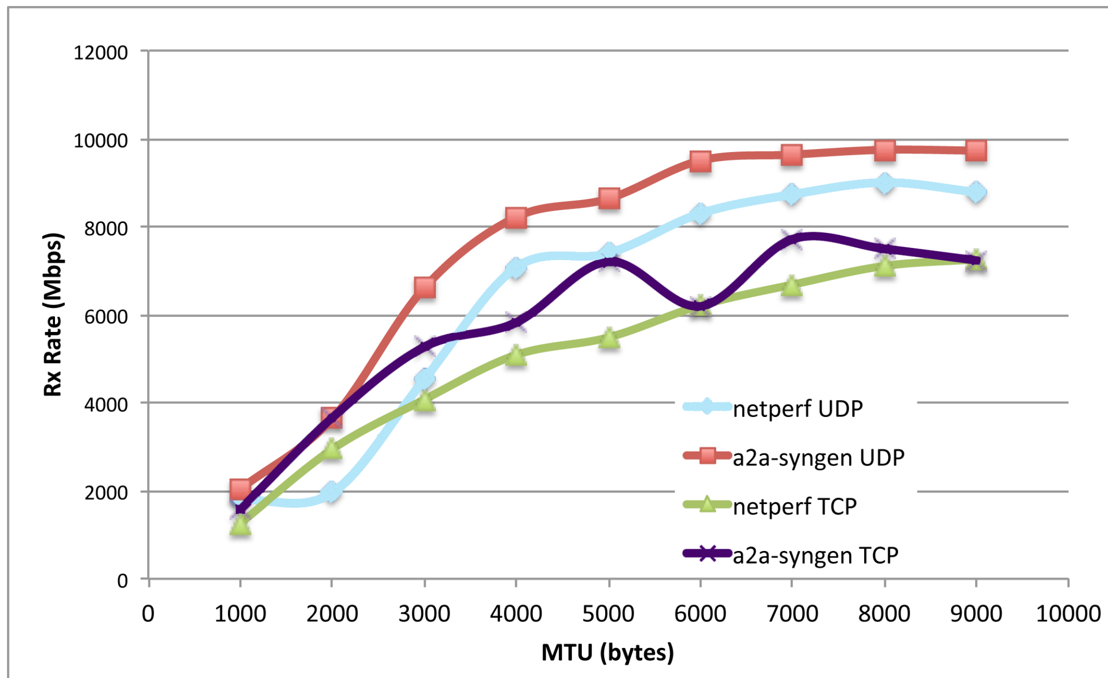


Figure 4.15. Throughput comparison between a2a-syngen and netperf for varying MTU between two hosts

process, and send an acknowledgement. Then, S sends a command for hosts to enter the *SETUP-CLIENT* phase, during which, the client processes start and connect to the N-1 remote host server clients. Each host may optionally add extra instrumentation (e.g. tcpdump) depending on experimental configuration. After receiving all acknowledgments, S sends the command for each host to run the experiment. When a host finishes its part of the experiment, it sends an acknowledgment. When S receives all acknowledgments, the final command issues to tell each host to enter the *CLEANUP* phase.

Figure 4.15 compares the throughput of a2a-syngen and netperf [53] for a two host experiment as the available MTU increases from 1000 to 9000 bytes.[11] The figure shows a2a-syngen delivers higher throughput than netperf for any MTU size for both UDP and TCP protocols. Further, a2a-syngen achieves 98.0% utilization of the 10 Gb/s NIC as the MTU approaches 9000 bytes. As the number of hosts increases to 23 (not

[11]We use only two hosts due to the limitations of netperf

shown in Figure 4.15), a2a-syngen appears to fairly divide its bandwidth between each remote host with only 0.14% standard deviation from the mean bandwidth. However, we observe that as more hosts participate, the aggregate bandwidth of all hosts increases, while the average receive rate of each individual host decreases. This issue relates to the OS overhead of processing multiple TCP/IP connections simultaneously, and we discuss it further in Section 4.5.2.

4.5 Evaluation

There are two questions that this evaluation seeks to answer. The first question is if a software TOR can keep up with the 11.5 μs switching time of the Mordia OCS. In order to achieve this, the overhead of OS packet control must be on the order of a microsecond. Otherwise, the software TOR would likely transmit packets during circuit reconfiguration or to the wrong destination host. The second question is if the software TOR can support a TCP/IP protocol near the performance seen on the EPS. If both of these goals can be met, then it is possible to measure the performance of all-to-all traffic on the prototype, and to run data center applications.

4.5.1 Emulated TOR Software

Figure 4.16 captures Host 1 receiving 8,966 octet UDP packets from Hosts 12–16 via the Qdisc described in Section 4.4.2 for a day and night of 123.5 μs and 11.5 μs, respectively. The transmission time and sequence number of each packet is determined from a *tcpdump* trace that runs on only Host 1.

First, we note that it takes on average 33.2 μs for the first packet of each circuit to reach Host 1. The *tcpdump* trace indicates that it takes less than 1 μs on average to process the synchronization frame and transmit the first packet. When sending a 9000-octet frame (7.2 μs), the packet spends at least 23 μs in the NIC before being

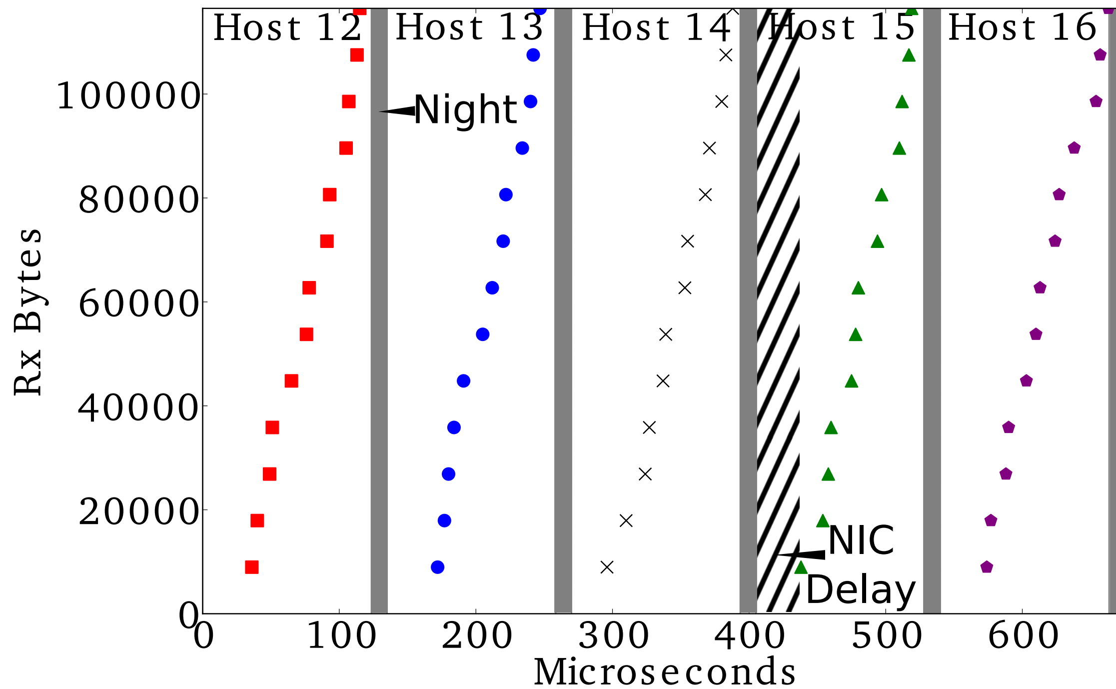


Figure 4.16. Host 1's Qdisc receiving UDP packets from Hosts 12–16 as it cycles through circuits connecting it to 22 other hosts.

transmitted. To prevent this *NIC delay* from causing packet loss in excess of 0.5%, the OS Qdisc must stop queuing packets for transmission 23 μs early. The result is that the Qdisc cannot use 23 μs of the circuit's day due to the behavior of the underlying NIC hardware. Vattikonda et al. [112] have shown that NIC priority flow control (PFC) pause frames can be used to enable fine-grained scheduling of circuits on (all-electrical) data center networks, and this technique may be applied to Mordia.

Figure 4.17 shows that Host 1's Qdisc transmits the first packet of each circuit within 1-3μs of the synchronization event that announces the new circuit. All packets for a circuit transmit in an average of 36 μs measured from day's beginning, which indicates that the processor is able to queue packets to the NIC faster than it can transmit them. It is also necessary for the Qdisc to stop transmitting 23.5 μs to keep average loss rate below 0.5%, because of delay in the NIC transmission. These 23.5 μs reduce the maximum duty possible for a fixed day and night, as no packets may be transmitted during them.

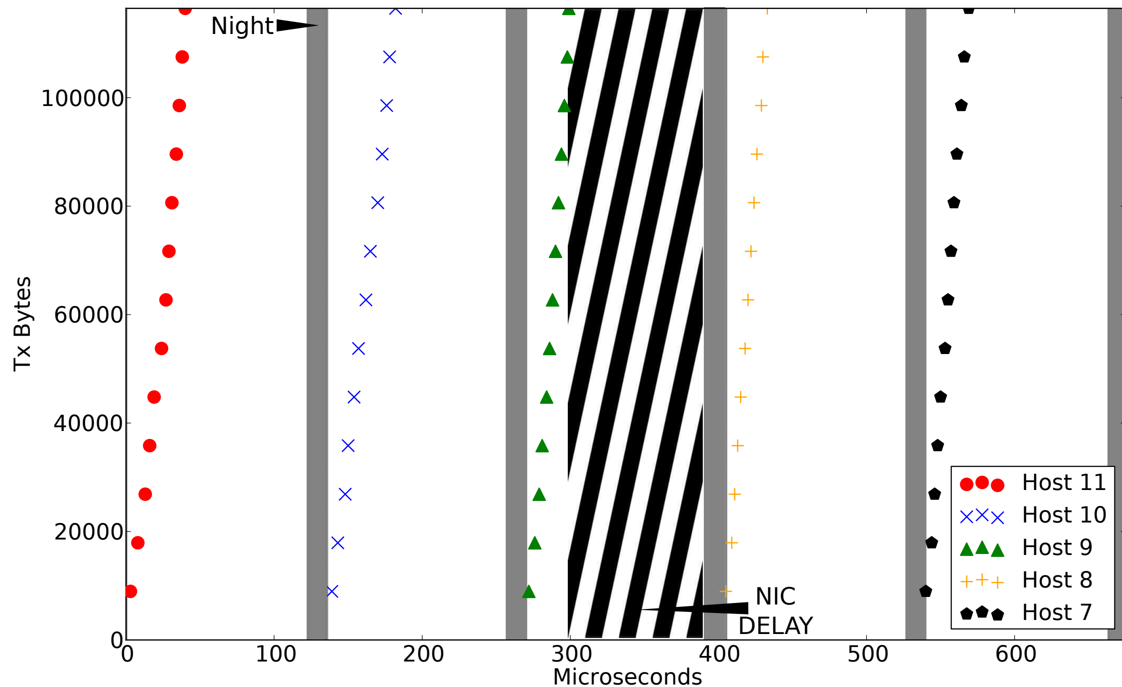


Figure 4.17. Host 1's Qdisc transmitting UDP packets to Hosts 7–11 as it cycles through circuits connecting it to 22 other hosts.

Figure 4.16 represents these two issues as *NIC DELAY*.

To summarize, the Linux hosts acting as emulated TORs are able to drain packets into the network during the appropriate "day," which can be as small as 61 μs. However, jitter in receiving synchronization packets results in a 0.5% overall loss rate, and there is a 23 μs delay after each switch reconfiguration before the NIC begins sending packets to the OCS. These overheads are specific to our use of commodity hosts as TORs.

4.5.2 Throughput

We generate all-to-all TCP and UDP traffic (506 GB) between 23 hosts and measure the throughput both over a traditional electrical packet switch (EPS, used as a baseline) as well as our Mordia OCS prototype including emulated TOR switches. Results are shown in Figure 4.18. The throughput over the EPS serves as an upper bound on the potential performance over the OCS. In addition, throughput over the OCS is

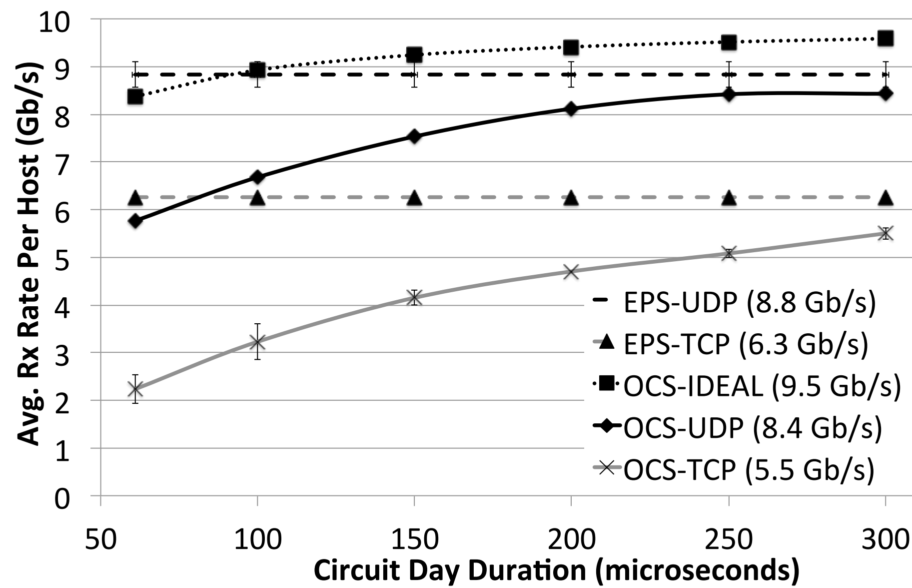


Figure 4.18. Throughput delivered over the OCS. The fundamental difference between ideal and observed is due to the OCS duty cycle. Further deviations are due to our system artifacts, namely the lack of segmentation offloading in the NIC, NIC-induced delay, and synchronization packet jitter. The minimum circuit day length is 61 μs. The legend shows the maximum average receive rate for each switch and protocol combination.

fundamentally limited by the OCS duty cycle.

Starting from the baseline, the EPS can deliver UDP traffic (EPS-UDP) at an average receive rate of 8.83 Gb/s to each of the 23 hosts. We find that as the number of hosts increases from 2 to 23, the average receive rate degrades from 9.81 Gb/s, which we attribute to the kernel and NIC. The EPS is capable of delivering TCP traffic at 8.69 Gb/s, which is within 1.6% of UDP traffic. However, this throughout relies on TCP segmentation offloading (TSO) support in the NIC, which is incompatible with our Mordia kernel module. The happens because, when circuits are reconfigured, any packets in flight are *runted* by the link going down, and we lose control over the transmission of packets when relying on TSO. Consequently, Mordia requires disabling TSO support. On an all-electrical packet network, TCP without TSO support is limited to 6.26 Gb/s (EPS-TCP), which we use as an upper bound on the performance we expect to see over

the OCS.

Figure 4.18 shows the raw bandwidth available to each host (calculated as the duty cycle) from the OCS as OCS-IDEAL. It is important to remember that this line does not account for the 23.5 μs *NIC delay* which acts to reduce measured duty cycle even more. For the experiments, we varied the OCS day duration between 61–300 μs to observe the effect of different duty cycles.[12] The OCS's UDP throughput (OCS-UDP) ranges from 5.726-8.429 Gb/s, or within 4.6% of EPS-UDP. The major reasons for the discrepancy are duty cycle, NIC delay, the OS's delay in handling a softirq, and synchronization jitter (see discussion below).

TCP throughput on the OCS (OCS-TCP) ranges from 2.231–5.501 Gb/s, or within 12.1% of EPS-TCP for large day durations. TCP throughput has all of the same issues as UDP throughput, in addition to two new problems. First, TCP traffic cannot use TSO to offload, and so the TCP/IP stack becomes CPU-bound handling the required 506 connections. Second, the observed 0.5% loss rate invokes congestion control, which decreases throughput. However, TCP does show an upward trend in bandwidth with increasing duty cycle. We attempted the kernel bypass techniques from Vattikonda et al. [112] to eliminate NIC delay and the softirq variance, but found that it was difficult to minimize loss rate below 5% due to interruptions from the OS that would cause packets to be sent to the wrong host during reconfiguration.

We mention above that synchronization jitter causes a reduction in throughput, and is also responsible for packet loss. Synchronization packets are generated in hardware to minimize latency and jitter, but the OS can add jitter in how its devices receive the packets and schedule softirqs. To measure this jitter, we set the day and night to 106 μs and 6 μs, respectively, and capture 1,922,507 synchronization packets across 3 random hosts. We compute the difference in timestamps between each packet and expect to

[12] Due to the programming time of the WSSs, the smallest day duration we support is 61 μs

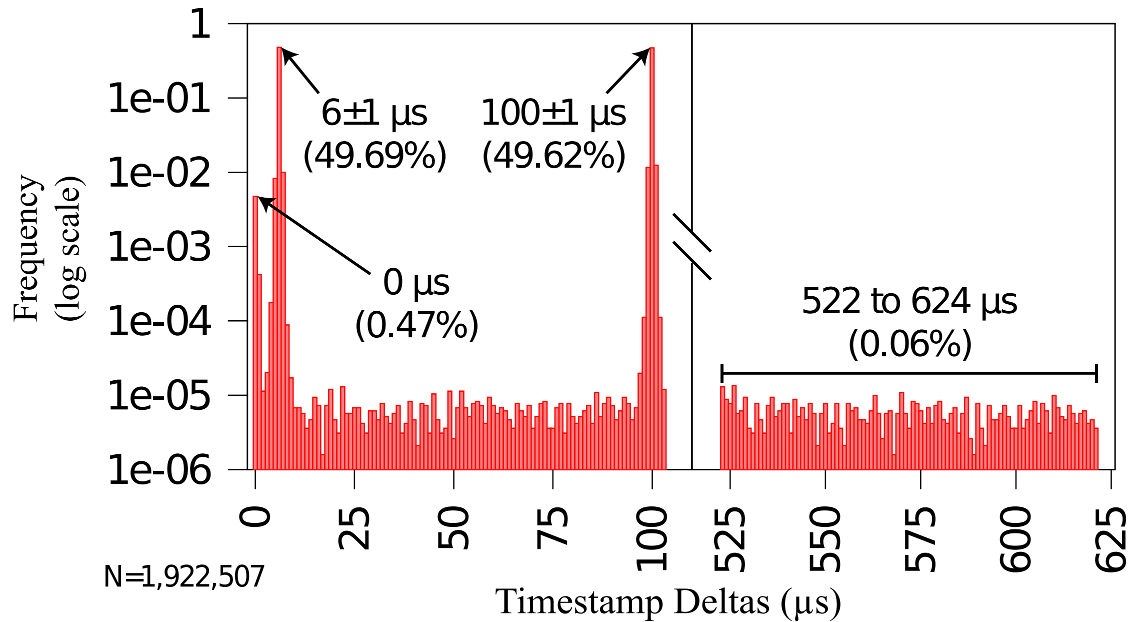


Figure 4.19. Synchronization jitter as seen by our software TOR's OS.

see packets arriving with timestamp deltas of either 6±1 μs or 100±1 μs. We found that 99.31% of synchronization packets arrive at their expected times, 0.47% of packets arrive with timestamp deltas of zero, and 0.06% packets arrive with timestamp deltas between 522 μs and 624 μs (see Figure 4.19). The remaining 0.16% of packets arrive between 7–99 μs. We also point out that the 0.53% (0.47% + 0.06%) of synchronization packets that arrive at unexpected times is very close to our measured loss rate. Our attempts to detect bad synchronization events in the Qdisc did not change the loss rate measurably. Firmware changes in the NIC could be used to entirely avoid the need for these synchronization packets by directly measuring the link up/down events.

To summarize, despite the non-realtime behavior inherent in emulating TORs with commodity PCs, we are able to achieve 95.4% of the bandwidth of a comparable EPS with UDP traffic, and 87.9% of an EPS, sending non-TSO TCP traffic. We are encouraged by these results, which we consider to be lower bounds of what would be possible with more precise control over the TOR.

4.6 Summary

Network bandwith demand is likely to increase from software techniques that leverage memory for storage, improved disk technologies, and faster server NICs. Hence the network is a very likely source of inefficiency for future data center applications, causing both increased energy consumption and straining QoS agreements. Optics has the potential to improve data center bandwidth by an order of magnitude, while also allowing the switch to use two orders of magnitude less power per port. This chapter's analysis indicates that an OCS network can reduce data center power by 24.7%, if the network needs to provide full bisection bandwidth at 100 Gb/s to non-energy proportional servers.

We follow this analysis with the design of the Mordia OCS architecture, and evaluate it on a 24-port prototype. A key contribution of this chapter is how to modify a commodity OS to act as a microsecond TOR, capable of supporting a traditional TCP/IP stack on top of an OCS prototype that reduces switching time by 3 orders of magnitude. The software TOR is capable of reacting within a microsecond to the reconfiguration of the Mordia OCS. Further, the microsecond TOR demonstrates that UDP and TCP protocols can achieve up to 95.4% and 87.9% of bandwidth offered by the OCS prototype, enabling data center applications to run more efficiently on a hybrid electrical/optical network.

4.7 Acknowledgments

Chapter 4 was supported by the National Science Foundation through NSF MRI grant CNS-0923523 and through CIAN NSF ERC under grant EEC-0812072. Portions of this chapter were funded through a Google Focused Research Award. Electrical packet switches were provided through a donation from Cisco Systems. Chapter 4 was also

supported by Microsoft and Ericsson.

Chapter 4 contains material from “Integrating Microsecond Circuit Switching into the Data Center”, by George Porter, Richard Strong, Nathan Farrington, Alex Forencich, Pang-Chen Sun, Tajana S. Rosing, Yeshaiahu Fainman, George Papen, and Amin Vahdat, which will appear in the proceedings of *The Special Interest Group on Data Communications*, 2013. The dissertation author was the secondary investigator and author of this paper.

Chapter 5

Summary

Demand for cloud services, big data, and the data centers that support them will likely grow for the foreseeable future. Today, a single data center can draw as much as 100 MW. A major source of this power draw is the servers, which lack energy proportionality. Even when a server is fully utilized as measured by CPU utilization, it may still not be executing efficiently. Moore's Law will not resolve the problem on its own, demanding new hardware and software techniques.

For these reasons, this dissertation presents low-latency techniques that use hardware and/or software to improve the energy efficiency of data centers. Chapter 2 explores a technique that combines power gating, slave latches, and source biasing. The result is a core that can wake up from a power-gated state in as little as 8.06 ns, while being able to retain critical state. We apply power gating to cores stalled on a memory access. We propose MAPG-Counter, which employs a predictive scheme to estimate the duration of memory accesses. We also propose TAP that uses tokens transmitted on the cache interconnect to avoid any performance overhead. This technique tracks the lower-bound duration of each access to allow cores to wake up just before they are needed. For applications that stall often from accessing the memory subsystem, TAP achieves up to 22.4% energy savings for out-of-order cores. As the number of cores trying to access the memory subsystem at once increases from 1 to 32, TAP can detect the longer

duration of core stalls, and power gate the core 3.7× longer. However, MAPG-Counter can save more energy on average (4.1%) for in-order cores. In addition, both TAP and MAPG-Counter scale to many core designs through the introduction of wake-up stagger and wake-up slots.

Chapter 3 aims to reduce the latency of core switching to enable improvements in techniques that rely on it. Such techniques include software data spreading [66], load-balancing, VM migration, and fast thread switching between asymmetric cores. Specifically, we cut Linux core-switching latencies in half or better, by avoiding extra context switches, unnecessary scheduler calculations, the long code latency of IPI interrupts, and by waking up target cores early. We show core switching has no performance impact on macrobenchmarks for a real server. Simulations show that core switching can sometimes reduce and sometimes improve performance for macrobenchmarks, but yet can offer energy savings of 3.37× on average for asymmetric hardware.

Techniques like those in Chapters 2 and 3 act to improve the energy proportionality of server processors, but assume that the network can always provide the server with data to process. Contrary to this view, data center networks are often oversubscribed and network bandwidth demands will likely grow in the near future. Optics can increase data center network bandwidth by an order of magnitude, while reducing the power per port of switch infrastructure by two orders of magnitude. However, current OCS prototypes switch circuits instead of packets, which raises concerns over whether an OCS can switch fast enough to support all data center traffic patterns.

Chapter 4 discusses the experience of integrating a prototype, microsecond OCS into the data center. The OCS can reconfigure circuits in 11.5 μs, three orders of magnitude faster that previous work [43, 115]. In order to support system level experiments, Chapter 4 shows how to modify a commodity operating system to respond to circuit reconfiguration within a microsecond. The modifications still support a traditional TCP/IP

stack, allowing us to demonstrate up to 95.4% UDP and 87.9% TCP throughput across the OCS prototype. Further, the chapter analytically shows that an OCS could reduce data center power by 24.7% compared to an EPS at 100 Gb/s per host with non-energy proportional servers.

Although this dissertation contributes to the energy efficiency of the data center for both the server processor and data center networks, the problem is still far from solved. There are several questions related to the techniques of Chapters 2, 3, and 4. We now discuss some of these questions that are left for future work.

In the case of server CPU-level power gating, one big question is whether it is necessary to avoid any performance overhead. TAP currently wakes up the core for the soonest memory response. However, should the core stall while more than one memory access is ongoing, it is quite possible that more energy would be saved by waiting for the last memory response. The reasoning is that not many instructions can execute if an instructional dependency exists for the last memory response; in comparison, a core with all memory requests satisfied may achieve higher instructional level parallelism. In addition, TAP is extremely conservative about its power-gating intervals, while MAPG-Counter is willing to trade a small performance hit for the chance of better energy savings. A technique that switches between the two strategies may save more energy on a larger variety of architectures. Functional unit power gating is a complimentary technique to both TAP and MAPG-Counter. Cooperation between functional unit and memory access power gating would likely result in higher overall energy savings. In the case of a communication intensive parallel benchmark, Chapter 2 leaves open the question of how best to allow core power gating to coexist with frequent coherency actions. Currently, we always source bias the core's instruction and data cache, and wake it for any coherency action. It may be better to leave on the instruction and data cache if a coherence action is likely, as the result is lower communication delay and wake-up energy.

Very fast core switching from Chapter 3 also raises numerous questions. When core switching on an asymmetric multicore, it is not clear which core best suits the OS specific code. A study is necessary to test core switching on various core architectures in order to find out which components most correlate with good performance. Once a good OS core-switching architecture is found, the issue of cache state migration arises. Chapter 3 avoids this issue by assuming frequent system calls, which can keep the cache warm if it is reasonably sized. However, if we combine fast core switching with a technique that predicts and migrates the cache working set [22], core switching could be applicable to more situations. A clear follow on question would be how to add cache working set prediction and migration to the OS with minimal hardware support. This could lead to even lower latencies for thread migration in today's servers.

Finally, Chapter 4 delivers an OCS prototype that can reconfigure circuits in 11.5 μs, is compatible with a traditional Linux TCP/IP stack, and can run system level experiments. The immediate question is how will different data center applications behave on this prototype. To answer this question, it will also be necessary to implement an OCS scheduler that can configure the OCS to offer circuits most needed each 100 μs or less. If the OCS cannot support all types of applications, then it would be worthwhile to consider what should change about the OCS to broaden the number of supported applications.

In conclusion, this dissertation presents several low-latency techniques that can improve the energy efficiency of the data center. We believe that low-latency tecniques are critical to this effort so as to avoid impacting the QoS that users demand from their data center providers. Further, we have shown that there is significant room in both the hardware and software domain to address data center energy efficiency challenges. It is the hope of this author that efforts on both the hardware and software front will continue until the entire data center becomes energy proprotional.

Bibliography

[1] Cisco Data Center Infrastructure 2.5 Design Guide. http://www.cisco.com/en/US/docs/solutions/Enterprise/Data_Center/DC_Infra2_5/DCI_SRND_2_5a_book.html.

[2] CloudEngine 12800 Series High-Performance Core Switches. http://enterprise.huawei.com/us/products/network/switch/data-center-switch/hw-133602.htm.

[3] Glimmerglass 80x80 MEMS Switch. http://www.glimmerglass.com/products/technology/.

[4] Hadoop: Open Source Implementation of Map Reduce. http://hadoop.apache.org/.

[5] EPA Report on Server and Data Center Energy Efficiency. U.S. Environmental Protection Agency, Energy Star Program, 2007.

[6] International Technology Roadmap for Semiconductors. http://www.itrs.net/Links/2010ITRS/2010Update, 2010.

[7] Myricom Sniffer10G – Cybersecurity Software. https://www.myricom.com/sniffer.html, 2011.

[8] Thinkmate RAX QS3-4110. http://www.thinkmate.com/System/RAX_QS3-4110, 2013.

[9] J. Aas. Understanding the Linux 2.6.8.1 CPU Scheduler. http://josh.trancesoftware.com/linux/, 2005.

[10] D. Abts, M. R. Marty, P. M. Wells, P. Klausler, and H. Liu. Energy Proportional Datacenter Networks. In *Proc. International Symposium on Computer Architecture*, pages 338–347, 2010.

[11] K. Agarwal, K. Nowka, H. Deogun, and D. Sylvester. Power Gating with Multiple Sleep Modes. In *Proc. International Symposium on Quality Electronic Design*,

pages 633–637, 2006.

[12] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity, Data Center Network Architecture. In *Proc. Special Interest Group on Data Communications*, 2008.

[13] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *Proc. USENIX Conference on Networked Systems Design and Implementation*, pages 19–19, 2010.

[14] M. Annavaram. A Case for Guarded Power Gating for Multi-Core Processors. In *Proc. International Symposium on High Performance Computer Architecture*, pages 291–300, 2011.

[15] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai. The Impact of Performance Asymmetry in Emerging Multicore Architectures. In *Proc. International Symposium on Computer Architecture*, pages 506–517, 2005.

[16] L. A. Barroso and A. Hölzle. The Case for Energy-Proportional Computing. *IEEE Computer*, 40(12):33–37, December 2007.

[17] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel. Finding a Needle in Haystack: Facebook's Photo Storage. In *Proc. USENIX Conference on Operating Systems Design and Implementation*, pages 1–8, 2010.

[18] M. Becchi and P. Crowley. Dynamic Thread Assignment on Heterogeneous Multiprocessor Architectures. *Journal on Instruction-Level Parallelism*, pages 1–26, 2008.

[19] C. Benvenuti. *Understanding Linux Network Internals*. O'Reilly Media, Inc., 2006.

[20] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The M5 Simulator: Modeling Networked Systems. *IEEE Micro*, 26(4):52–60, 2006.

[21] G. Birkhoff. Tres Observaciones Sobre el Algebra Lineal. *Universidad Nacional de Tucuman Revista*, 5:147–151, 1946.

[22] J. A. Brown, L. Porter, and D. M. Tullsen. Fast Thread Migration via Cache Working Set Prediction. In *Proc. International Conference on High-Performance*

Computer Architecture, pages 193–204, 2011.

[23] J. A. Brown and D. M. Tullsen. The Shared-Thread Multiprocessor. In *Proc. International Conference on Supercomputing*, pages 73–82, 2008.

[24] J. Brutlag. Speed Matters for Google Web Search. http://services.google.com/fh/files/blogs/google_delayexp.pdf, 2009.

[25] C. Castellanos. PCI-SIG Announces PCI EXPRESS 4.0 Evolution to 16GT/S, Twice the Throughput of PCI EXPRESS 3.0 Technology. http://www.pcisig.com/news_room/Press_Releases/PCIe_4_0_BitRate_Release_11_29_11.pdf, 2011.

[26] A. M. Caulfield, A. De, J. Coburn, T. I. Mollow, R. K. Gupta, and S. Swanson. Moneta: A High-Performance Storage Array Architecture for Next-Generation, Non-Volatile Memories. In *Proc. International Symposium on Microarchitecture*, pages 385–395, 2010.

[27] K. Chakraborty, P. M. Wells, and G. S. Sohi. Computation Spreading: Employing Hardware Migration to Specialize CMP Cores On-the-fly. In *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[28] P. Chaparro, J. Gonzalez, and A. Gonzalez. Thermal-Aware Clustered Microarchitectures. In *Proc. International Conference on Computer Design*, pages 48–53, 2004.

[29] K. Chen, A. Singlay, A. Singhz, K. Ramachandranz, L. Xuz, Y. Zhangz, X. Wen, and Y. Chen. OSA: An Optical Switching Architecture for Data Center Networks with Unprecedented Flexibility. In *Proc. USENIX Conference on Networked Systems Design and Implementation*, pages 18–18, 2012.

[30] M. Cho, N. Sathe, M. Gupta, S. Kumar, S. Yalamanchilli, and S. Mukhopadhyay. Proactive Power Migration to Reduce Maximum Value and Spatiotemporal Non-Uniformity of On-Chip Temperature Distribution in Homogeneous Many-Core Processors. In *Proc. Semiconductor Thermal Measurement and Management Symposium*, pages 180–186, 2010.

[31] B. Choi, L. Porter, and D. M. Tullsen. Accurate Branch Prediction for Short Threads. In *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[32] M. H. Chowdhury, J. Gjanci, and P. Khaled. Innovative Power Gating for Leakage

Reduction. In *Proc. International Symposium on Circuits and Systems*, pages 1568–1571, 2008.

[33] T. Constantinou, Y Sazeides, P. Michaud, D. Fetis, and A. Seznec. Performance Implications of Single Thread Migration on a Chip Multi-Core. In *Workshop on Design, Architecture, and Simulation of Chip Multiprocessors*, 2005.

[34] G. Cook and J. Horn. How Dirty Is Your Data? A Look at the Energy Choices that Power Cloud Computing. Greenpeace International, 2011.

[35] A. K. Coskun, R. Strong, D. M. Tullsen, and T. S. Rosing. Evaluating the Impact of Job Scheduling and Power Management on Processor Lifetime for Chip Multiprocessors. In *Proc. Conference on Measurement and Modeling of Computer Systems*, pages 169–180, 2009.

[36] M. DeVuyst, A. Venkat, and D. M. Tullsen. Execution Migration in a Heterogeneous-ISA Chip Multiprocessor. In *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 261–272, 2012.

[37] G. Dhiman, K. Pusukuri, and T. Rosing. Analysis of Dynamic Voltage Scaling for System Level Energy Management. In *Proc. Conference on Power Aware Computing and Systems*, pages 9–9, 2008.

[38] G. Dhiman and T. Rosing. Dynamic Voltage Frequency Scaling for Multi-Tasking Systems Using Online Learning. In *Proc. International Symposium on Low Power Electronics and Design*, pages 207–212, 2007.

[39] S. Dobre. Qualcomm CDMA Technologies, Inc. *Personal communication*, September 2011.

[40] X. Fan, W. D. Weber, and L. A. Barroso. Power Provisioning for a Warehouse-Sized Computer. In *Proc. International Symposium on Computer Architecture*, pages 13–23, 2007.

[41] N. Farrington. *Optics in Data Center Network Architecture*. PhD thesis, University of California at San Diego, 2013.

[42] N. Farrington, G. Porter, Y. Fainman, G. Papen, and A. Vahdat. Hunting Mice with Microsecond Circuit Switches. In *Proc. Workshop on Hot Topics in Networks*, pages 115–120, 2012.

[43] N. Farrington, G. Porter, S. Radhakrishnan, H. H. Bazzaz, V. Subramanya, Y. Fainman, G. Papen, and A. Vahdat. Helios: A Hybrid Electrical/Optical Switch Architecture for Modular Data Centers. In *Proc. Special Interest Group on Data Communications*, pages 339–350, 2010.

[44] A. Fedorova. *Personal communication*, 2009.

[45] A. Fedorova, D. Vengerov, and D. Doucette. Operating System Scheduling On Heterogeneous Core Systems. In *Proc. Workshop on Operating Systems Support for Heterogeneous Multicore Architectures*, 2007.

[46] B. Fitzpatrick. Distributed Caching with Memcached. *Linux Journal*, 2004(124):5–5, 2004.

[47] J. E. Ford, V. A. Aksyuk, D. J. Bishop, and J. A.Walker. Wavelength Add-Drop Switching using Tilting Micromirrors. *IEEE Journal of Lightwave Technology*, 17:904–911, 1999.

[48] R. E. Grant and A. Afsahi. Power-Performance Efficiency of Asymmetric Multiprocessors for Multi-Threaded Scientific Applications. In *Proc. International Parallel and Distributed Processing Symposium*, April 2006.

[49] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A Scalable and Flexible Data Center Network. In *Proc. Special Interest Group on Data Communications*, pages 51–62, 2009.

[50] D. Halperin, S. Kandula, J. Padhye, P. Bahl, and D. Wetherall. Augmenting Data Center Networks with Multi-Gigabit Wireless Links. In *Proc. Special Interest Group on Data Communications*, 2011.

[51] K. He, R. Luo, and Y. Wang. A Power Gating Scheme for Ground Bounce Reduction During Mode Transition. In *Proc. International Conference on Computer Design*, pages 388–394, 2007.

[52] S. Heo, K. Barr, and K. Asanovic. Reducing Power Density through Activity Migration. In *Proc. International Symposium on Low Power Electronic Design*, 2003.

[53] Hewlett-Packard Company. Netperf: A Network Performance Benchmark. http://www.netperf.org.

[54] M. D. Hill and M. R. Marty. Amdahl's Law in the Multicore Era. *IEEE Computer*,

41(7):33–38, July 2008.

[55] U. Hoelzle and L. A. Barroso. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 1st edition, 2009.

[56] M. Horiguchi, T. Sakata, and K. Itoh. Switched-Source-Impedance CMOS Circuit for Low Standby Subthreshold Current Giga-Scale LSI's. *IEEE Journal of Solid-State Circuits*, 28(11):1131–1135, November 1993.

[57] L. R. Hsu, A. G. Saidi, N. L. Binkert, and S. K. Reinhardt. Sampling and Stability in TCP/IP Workloads. In *Proc. Workshop on Modeling, Benchmarking, and Simulation*, 2005.

[58] Z. Hu, A. Buyuktosunoglu, V. Srinivasan, V. Zyuban, H. Jacobson, and P. Bose. Microarchitectural Techniques for Power Gating of Execution Units. In *Proc. International Symposium on Low Power Electronics and Design*, pages 32–37, 2004.

[59] G. Huang, D. Sekar, A. Naeemi, K. Shakeri, and J. D. Meindl. Physical Model for Power Supply Noise and Chip/Package Co-Design in Gigascale Systems with the Consideration of Hot Spots. In *Proc. Custom Integrated Circuits Conference*, pages 841–844, 2007.

[60] B. Hubert, T. Graf, G. Maxwell, R. van Mook, M. van Oosterhout, P. B. Schroeder, J. Spaans, and P. Larroy. Linux Advanced Routing & Traffic Control HOWTO. http://www.lartc.org/lartc.html, 2002.

[61] Intel Corp. Intel Core 2 Duo Processors and Intel Core 2 Extreme Processors on 45-nm Process: Datasheet. Document Number 320120, July 2008.

[62] C. Isci, A. Buyuktosunoglu, C.-Y. Chen, P. Bose, and M. Martonosi. An Analysis of Efficient Multi-Core Global Power Management Policies: Maximizing Performance for a Given Power Budget. In *Proc. International Symposium on Microarchitecture*, pages 347–358, 2006.

[63] C. Isci, A. Buyuktosunoglu, and M. Martonosi. Long-Term Workload Phases: Duration Predictions and Applications to DVFS. *IEEE Micro*, 25(5):39–51, 2005.

[64] N. James, P. Restle, J. Friedrich, B. Huott, and B. McCredie. Comparison of Split-Versus Connected-Core Supplies in the POWER6 Microprocessor. In *Proc. International Solid-State Circuits Conference*, pages 298–604, 2007.

[65] K. Jeong, A. B. Kahng, S. Kang, T. S. Rosing, and R. Strong. MAPG: Memory Access Power Gating. In *Proc. Design, Automation, & Test in Europe Conference & Exhibition*, pages 1054–1059, 2012.

[66] M. Kamruzzaman, S. Swanson, and D. M. Tullsen. Software Data Spreading: Leveraging Distributed Caches to Improve Single Thread Performance. *SIGPLAN Conference on Programming Language Design and Implementation*, 45(6):460–470, 2010.

[67] M. Keating, D. Flynn, R. Aitken, A. Gibbons, and K. Shi. *Low Power Methodology Manual: For System-on-Chip Design*. Springer Publishing Company, Inc., 2007.

[68] N. S. Kim, J. Seomun, A. Sinkar, J. Lee, T. H. Han, K. Choi, and Y. Shin. Frequency and Yield Optimization Using Power Gates in Power-Constrained Designs. In *Proc. International Symposium on Low Power Electronics and Design*, pages 121–126, 2009.

[69] S. Kim, S. V. Kosonocky, and D. R. Knebel. Understanding and Minimizing Ground Bounce During Mode Transition of Power Gating Structures. In *Proc. International Symposium on Low Power Electronics and Design*, pages 22–25, August.

[70] S. Kim, S. V. Kosonocky, D. R. Knebel, K. Stawiasz, and M. C. Papaefthymiou. A Multi-Mode Power Gating Structure for Low-Voltage Deep-Submicron CMOS ICs. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 54(7):586–590, 2007.

[71] A. V. Krishnamoorthy, R. Ho, X. Zheng, H. Schwetman, J. Lexau, P. Koka, Guoliang L., I. Shubin, and J. E. Cunningham. Computer Systems Based on Silicon Photonic Interconnects. *IEEE*, 97(7):1337–1361, 2009.

[72] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen. Single-ISA Heterogeneous Multi-core Architectures: The Potential for Processor Power Reduction. In *Proc. International Symposium on Microarchitecture*, 2003.

[73] R. Kumar and G. Hinton. A family of 45nm IA processors. In *Proc. International Solid-State Circuits Conference - Digest of Technical Papers*, pages 58–59, 2009.

[74] R. Kumar, D. Tullsen, P. Ranganathan, N. Jouppi, and K. Farkas. Single-ISA Heterogeneous Multi-core Architectures for Multithreaded Workload Performance. In *Proc. International Symposium of Computer Architecture*, 2004.

[75] S. Kumar, C. J. Hughes, and A. Nguyen. Carbon: Architectural Support for Fine-Grained Parallelism on Chip Multiprocessors. In *Proc. International Symposium on Computer Architecture*, pages 162–173, 2007.

[76] J. Leverich, M. Monchiero, V. Talwar, P. Ranganathan, and C. Kozyrakis. Power Management of Datacenter Workloads Using Per-Core Power Gating. *IEEE Computer Architecture Letters*, 8(2):48–51, July 2009.

[77] H. Li, C. Cher, T. N. Vijaykumar, and K. Roy. VSV: L2-Miss-Driven Variable Supply-Voltage Scaling for Low Power. In *Proc. International Symposium on Microarchitecture*, pages 19–28, 2003.

[78] S. Li, J. Ahn, R. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *Proc. International Symposium on Microarchitecture*, pages 469–480, 2009.

[79] T. Li, D. Baumberger, D. A. Koufaty, and S. Hahn. Efficient Operating System Scheduling for Performance-Asymmetric Multi-Core Architectures. In *Proc. Supercomputing*, pages 1–11, 2007.

[80] T. Li, P. Brett, B. Hohlt, R. Knauerhase, S. D. McElderry, and S. Hahn. Operating System Support for Shared-ISA Asymmetric Multi-Core Architectures. In *Proc. Workshop on the Interaction between Operating Systems and Computer Architecture*, 2008.

[81] Y. Lin, C. Yang, J. Huang, and N. Chang. Memory Access Aware Power Gating for MPSoCs. In *Proc. Asia and South Pacific Design Automation Conference*, pages 121–126, 2012.

[82] R. Love. *Linux Kernel Development Second Edition*. Pearson Education, Inc., 2005.

[83] A. Lungu, P. Bose, A. Buyuktosunoglu, and D. J. Sorin. Dynamic Power Gating with Quality Guarantees. In *Proc. International Symposium on Low Power Electronics and Design*, pages 377–382, 2009.

[84] J. D. McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture Newsletter*, pages 19–25, 1995.

[85] R. McDougall and J. Mauro. *Solaris Internals: Solaris 10 and OpenSolaris Kernel*

Architecture. Prentice Hall, 2nd edition, 2007.

[86] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Computer Communications Review*, 38(2):69–74, 2008.

[87] D. A. B. Miller and H. M. Ozaktas. Limit to the Bit-Rate Capacity of Electrical Interconnects from the Aspect Ratio of the System Architecture. *Journal of Parallel and Distributed Computing*, 41(1):42–52, 1997.

[88] J. C. Mogul. TCP Offload Is a Dumb Idea Whose Time Has Come. In *Proc. Conference on Hot Topics in Operating Systems*, pages 5–5, 2003.

[89] J. C. Mogul, J. Mudigonda, N. Binkert, P. Ranganathan, and V. Talwar. Using Asymmetric Single-ISA CMPs to Save Energy on Operating Systems. *Micro*, 8(3):26–41, May-June 2008.

[90] I. Molnar. Modular Scheduler Core and Completely Fair Scheduler. http://kerneltrap.org/node/8059, April 2007.

[91] M. Monchiero. *Personal communication*, 2008.

[92] G. E. Moore. Cramming More Components onto Integrated Circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff. *IEEE Solid-State Circuits Society Newsletter*, 11(5):33–35, 2006.

[93] R. N. Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. PortLand: A Scalable Fault-Tolerant Layer 2 Data Center Network Fabric. In *Proc. Special Interest Group on Data Communications*, 2009.

[94] D. Nellans, R. Balasubramonian, and E. Brunvand. A Case for Increased Operating System Support in Chip Multi-Processors. In *Proc. of 2nd IBM Watson* $P = ac^2$ *Conference*, 2005.

[95] D. Nellans, R. Balasubramonian, and E. Brunvand. OS Execution on Multi-Cores: Is Out-Sourcing Worthwhile? *Operating System Review*, 43, April 2009.

[96] M. A. Olson, K. Bostic, and M. Seltzer. Berkeley DB. In *Proc. FREENIX Track, USENIX Annual Technology Conference*, pages 183–191, 1999.

[97] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières,

S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The Case for RAMClouds: Scalable High-Performance Storage Entirely in DRAM. *SIGOPS Operating Systems Review*, 43(4):92–105, 2010.

[98] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder. Using SimPoint for Accurate and Efficient Simulation. In *Proc. International Conference on Measurement and Modeling of Computer Systems*, pages 318–319, 2003.

[99] B. Pfaff, J. Pettit, T. Koponen, K. Amidon, M. Casado, and S. Shenker. e.a.: Extending Networking into the Virtualization Layer. In *Proc. Workshop on Hot Topics in Networks*, 2009.

[100] C. Qiao and M. Yoo. Optical Burst Switching (OBS) - A New Paradigm for an Optical Internet. *Journal of High Speed Networks*, 8(1):69, 1999.

[101] H. Qin, Y. Cao, D. Markovic, A. Vladimirescu, and J. Rabaey. SRAM Leakage Suppression by Minimizing Standby Supply Voltage. In *Proc. International Symposium on Quality Electronic Design*, pages 55–60, 2004.

[102] A. Rasmussen, G. Porter, M. Conley, H. V. Madhyastha, R. N. Mysore, A. Pucher, and A. Vahdat. TritonSort: A Balanced Large-Scale Sorting System. In *Proc. Conference on Networked Systems Design and Implementation*, pages 3–3, 2011.

[103] A. Rogers, D. Kaplan, E. Quinnell, and B. Kwan. The Core-C6 (CC6) Sleep State of the AMD Bobcat x86 Microprocessor. In *Proc. International Symposium on Low Power Electronics and Design*, pages 367–372, 2012.

[104] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. DRAMSim2: A Cycle Accurate Memory System Simulator. *Computer Architecture Letters*, 10(1):16–19, January-June 2011.

[105] G. Semeraro, G. Magklis, R. Balasubramonian, D. H. Albonesi, S. Dwarkadas, and M. L. Scott. Energy-Efficient Processor Design Using Multiple Clock Domains with Dynamic Voltage and Frequency Scaling. In *Proc. International Symposium on High-Performance Computer Architecture*, pages 29–40, 2002.

[106] Y. Shin, J. Seomun, K. Choi, and T. Sakurai. Power Gating: Circuits, Design Methodologies, and Best Practice for Standard-Cell VLSI Designs. *ACM Transactions on Design Automation of Electronic Systems*, 15(4):1–37, October 2010.

[107] H. Singh, K. Agarwal, D. Sylvester, and K. J. Nowka. Enhanced Leakage Reduction Techniques Using Intermediate Strength Power Gating. *IEEE Transactions*

on Very Large Scale Integration Systems,, 15(11):1215–1224, 2007.

[108] R. Sinkhorn. A Relationship Between Arbitrary Positive Matrices and Doubly Stochastic Matrices. *The Annals of Mathematical Statistics*, 35(2):876–879, 1964.

[109] J. M. Smith. A Survey of Process Migration Mechanisms. *ACM Operating System Review*, July 1998.

[110] T. A. Strasser and J. L. Wagener. Wavelength-Selective Switches for ROADM Applications. *IEEE Journal of Selected Topics in Quantum Electronics*, 16:1150–1157, 2010.

[111] J. S. Turner. Terabit Burst Switching. *Journal of High Speed Networks*, 8(1):3–16, 1999.

[112] B. C. Vattikonda, G. Porter, A. Vahdat, and A. C. Snoeren. Practical TDMA for Datacenter Ethernet. In *Proc. European Conference on Computer Systems*, pages 225–238, 2012.

[113] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor. Conservation Cores: Reducing the Energy of Mature Computations. *SIGARCH Computer Architecture News*, 38(1):205–218, 2010.

[114] J. von Neumann. A Certain Zero-Sum Two-Person Game Equivalent to the Optimal Assignment Problem. *Contributions to the Theory of Games*, 2:5–12, 1953.

[115] G. Wang, D. G. Andersen, M. Kaminsky, K. Papagiannaki, T. S. E. Ng, M. Kozuch, and M. Ryan. c-Through: Part-Time Optics in Data Centers. *SIGCOMM Computer Communication Review*, 40(4):327–338, 2010.

[116] O. Wechsler. Setting New Standards for Energy-Efficient Performance. Technology@Intel Magazine, 2006.

[117] B. Wun and P. Crowley. Network I/O Acceleration in Heterogeneous Multicore Processors. In *Proc. Symposium on High-Performance Interconnects*, pages 9–14, 2006.

[118] M. Zaharia. *Personal communication*, 2013.

[119] M. Zaharia, N. M. M. Chowdhury, M. Franklin, S. Shenker, and I. Stoica. Spark:

Cluster Computing with Working Sets. Technical Report, EECS Department, University of California, Berkeley, May 2010.

[120] K. Zhang, T. Zhang, and S. Pande. Binary Translation to Improve Energy Efficiency through Post-Pass Register Reallocation. In *Proc. International Conference on Embedded Software*, pages 74–85, 2004.

[121] L. Zhang, G. Dhiman, and T. S. Rosing. vGreenNet: Managing Server and Networking Resources of Co-Located Heterogeneous VMs. In *Proc. IPDPS High-Performance Grid and Cloud Computing Workshop*, 2013.

[122] Z. Zhang, X. Kavousianos, K. Chakrabarty, and Y. Tsiatouhas. A Robust and Reconfigurable Multi-Mode Power Gating Architecture. In *Proc. International Conference on VLSI Design*, pages 280–285, 2011.

[123] H. Zheng and Z. Zhu. Power and Performance Trade-Offs in Contemporary DRAM System Designs for Multicore Processors. *IEEE Transactions on Computers*, 59(8):1033–1046, August 2010.

[124] W. Zwaenepoel. Protocols for Large Data Transfers over Local Networks. In *Proc. Symposium on Data Communications*, pages 22–32, 1985.